UNIVERSIDADE FEDERAL DO RIO DE JANEIRO

CRISTIANO GURGEL DE CASTRO

EVINCED: ESQUEMA DE VERIFICAÇÃO DE INTEGRIDADE PARA SISTEMAS DE MEDIÇÃO BASEADOS EM CONTAGEM DE CICLOS E TEMPO

RIO DE JANEIRO
2017

Cristiano Gurgel de Castro

EVINCED: Esquema de verificação de integridade para sistemas de medição baseados em contagem de ciclos e tempo

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática, Instituto de Matemática e Instituto Tércio Parcitti da Universidade Federal do Rio de Janeiro(área de concentração Redes de Computadores e Sistemas Distribuídos), como parte dos requisitos necessários para a obtenção do Título de Mestre em Informática.

Orientador: Luiz Fernando Rust da Costa Carmo

Coorientador: Davidson Rodrigo Boccardo

Rio de Janeiro

2017

CIP - Catalogação na Publicação

C355e

Castro, Cristiano Gurgel de
EVINCED: Esquema de verificação de integridade para sistemas de medição baseados em contagem de ciclos e tempo / Cristiano Gurgel de Castro. -- Rio de Janeiro, 2017.
82 f.

Orientador: Luiz Fernando Rust da Costa Carmo.
Coorientador: Davidson Rodrigo Boccardo.
Dissertação (mestrado) - Universidade Federal do Rio de Janeiro, Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais, Programa de Pós-Graduação em informática, 2017.

1. verificação de integridade. 2. atestação de software. 3. segurança em sistemas embarcados. I. Carmo, Luiz Fernando Rust da Costa, orient. II. Boccardo, Davidson Rodrigo, coorient. III. Título.

Elaborado pelo Sistema de Geração Automática da UFRJ com os dados fornecidos pelo(a) autor(a).

EVINCED: Esquema de verificação de integridade para sistemas de medição baseados em contagem de ciclos e tempo

Cristiano Gurgel de Castro

Dissertação de Mestrado submetida ao Programa de Pós-graduação em Informática do Instituto de Matemática e do Instituo Tércio Pacitti da Universidade Federal do Rio de Janeiro - UFRJ, como parte dos requisitos necessários à obtenção do título de Mestre em Informática.

Aprovada em 28 de junho de 2017 por:

Prof. Dr. Luiz Fernando Rust da Costa Carmo, UFRJ (Presidente)

Prof. Dr. Davidson Rodrigo Boccardo, GreenHat

Prof. Dr. Claudio Miceli de Farias, UFRJ

Profa. Dra. Flávia Coimbra Delicato, UFRJ

Prof. Dr. Raphael Carlos Santos Machado, Inmetro

Este trabalho é dedicado a todos os amigos e colegas
que me acompanharam durante a difícil jornada.

AGRADECIMENTOS

A Deus.

Aos amigos e colegas do Inmetro.

Aos amigos e colegas da UFRJ, em especial aos do Labnet.

Aos meus orientadores Luiz Rust e Davidson Boccardo, pela sua atenção e paciência.

À Francisco, Tereza, Viviane e André, que com os quais, apesar de longe, sempre posso contar.

Aos demais membros da minha família, que tenho o prazer da convivência em toda a minha vida.

À Camila pela paciência e companheirismo nesses anos.

"Eu lhes garanto:
Vocês vão chorar e se lamentar,
mas o mundo vai se alegrar.
Vocês ficarão angustiados,
mas a angústia de vocês se transformará
em alegria"
(Bíblia Sagrada, Jo 16,20)

RESUMO

No presente trabalho apresenta-se um esquema de verificação de integridade de software para sistemas embarcados. A verificação de integridade é uma ferramenta de auditoria que permite a uma entidade Verificador checar se o software em execução em outra plataforma Disv não foi modificado de forma não autorizada. É útil para vários cenários do mundo real, em que tais modificações a dispositivos embarcados podem trazer diversos prejuízos financeiros, como em *smart grids*, e até mesmo riscos à vida, como no caso de software embarcados em automóveis por exemplo.

O esquema proposto visa a facilidade de implementação e é baseado em software. Por ser baseado em software, elimina a necessidade de hardware adicional à plataforma, o que é uma característica interessante para cenários de baixo custo, e também permite a aplicação em sistemas legados. O método aproveita-se de uma característica da plataforma de execução, especificamente, seus ciclos de *clock* de forma a compor o resultado da verificação de integridade.

Por fim, apresenta-se uma implementação de validação do esquema proposto. Ambas as entidades, Verificador e Disv, são implementados em plataformas amplamente disponíveis. Várias consultas de verificação de integridade foram obtidas através da interação entre as duas entidades. Os resultados obtidos apontam que o método proposto pode ser utilizado com sucesso em casos reais, já que obteve bons resultados na diferenciação entre um Disv íntegro e um Disv malicioso.

Palavras-chave: verificação de integridade. atestação de software. segurança em sistemas embarcados.

ABSTRACT

In this work, we present a software integrity verification scheme for embedded systems. Integrity Verification is an auditing tool that allows an entity, Verifier, to check if software running on another platform, Prover, has not been modified in an unauthorized manner. It is useful for many real-world scenarios, where such modifications to embedded devices can bring financial losses, for example in smart grids, and even life-threatening risks, such as in automotive software.

The proposed scheme aims at ease of implementation and is software based. Because it is software-based, it eliminates the need for additional hardware to the platform, which is an interesting feature for low-cost scenarios, and also enables its application in legacy systems. The method uses a characteristic of the execution platform, specifically, its clock cycles, in order to compose the result of the integrity verification.

Finally, a validation implementation of the proposed scheme is presented. Both Verifier and Prover entities are implemented on widely available platforms. Several integrity verification queries were obtained through the interaction between the two entities. The results obtained indicate that the proposed method can be used successfully in real cases, since it obtained good results in the differentiation between a untampered Prover and a malicious one.

Keywords: integrity verification. software attestation. security in embedded systems.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE SÍMBOLOS

$\|\|$	Operador de concatenação
bs	Tamanho de um dos blocos de memória, no qual é dividida a memória de programa do Disv para a execução da verificação de integridade.
$\mathcal{C}$	Desafio de verificação de integridade
D	Valor padrão das posições de memória não gravadas. Geralmente `0x00` ou `0xFF`
$\mathscr{M}$	Memória de programa de um Disv.
$\mathscr{M}_{\mathcal{P}}$	Memória de programa de um Disv íntegro
$\mathscr{M}_{\tilde{\mathcal{P}}}$	Memória de programa de um Disv malicioso
NEG	Resultado Negativo da verificação de integridade, indicando que o Disv é íntegro
N	Número de iterações (consultas à memória) realizadas no algoritmo de cálculo CalcAlg
POS	Resultado Positivo da verificação de integridade, indicando que o Disv é fraudulento
ps	Tamanho do prefixo, utilizado como critério de parada no desafio da verificação de integridade
$\mathcal{P}^{\checkmark}$	Disv íntegro
$\tilde{\mathcal{P}}$	Disv malicioso
$\mathcal{P}$	Disv. Entidade potencialmente maliciosa que responde à uma consulta de verificação de integridade
$\mathcal{R}$	Resposta do algoritmo de cálculo
$\mathcal{R}^{\checkmark}$	Resposta íntegro do algoritmo de cálculo
$\tilde{\mathcal{R}}$	Resposta maliciosa provinda de um Disv fraudulento.
s	Semente. Parâmetro aleatório que define o comportamento da rotina de verificação de integridade presente no Disv.
$\mathbf{S}$	Software embarcado no Disv
$\mathbf{S}$	Software íntegro embarcado no Disv

$\tilde{\mathbf{S}}$ Software malicioso embarcado no Disv

`sizeof` Função que retorna o tamanho do programa em bytes

$\mathcal{V}$ Verificador. Entidade confiável que inicia a verificação de integridade e julga o resultado provindo do Disv.

SUMÁRIO

1 INTRODUÇÃO

Dispositivos eletrônicos estão dispersos em diversos tipos de ambientes – industriais, residenciais, hospitalares, urbanos, pessoais, transporte, entre outros. Estes equipamentos cada vez mais poderosos e efetuando cálculos mais complexos e, por essa razão são adjetivados comumente de *smart*s. Também estamos na era da Internet das Coisas (CHUI; LÖFFLER; ROBERTS, 2010), em que vários dispositivos comuns utilizados, são dotados de sensores e/ou atuadores e podem se comunicar entre si, contribuindo para efetuar seu serviço de maneira mais aprimorada ou elaborada. Tais dispositivos podem coletar dados pessoais e sensíveis, tais como hábitos pessoais, hábitos de consumo, saúde, contatos, compromissos, localização, dados bancários, etc.

A segurança da informação tem sido evidenciada nos últimos anos devido a questões relacionadas a privacidade (LANGHEINRICH, 2001). Com a ampliação no número de dispositivos e da computação ubíqua, possíveis aberturas disponíveis em dispositivos e protocolos de comunicação podem ser exploradas de forma a permitir o tornar disponível dados pessoais e/ou corporativos sensíveis.

Tão importante quanto a privacidade de dados de usuários é a integridade de tais dispositivos. Estes equipamentos eletrônicos podem ser utilizados também em aplicações críticas (BANERJEE et al., 2012) como em votação eletrônica (WOLCHOK et al., 2010), aviônica, computadores de bordo em automóveis, marcapassos digitais e equipamentos hospitalares em geral, trocas comerciais. Muitos transtornos podem advir do funcionamento incorreto de tais equipamentos, já que uma falha em tais dispositivos pode então trazer danos à relações sociais e inclusive perigo direto à vida humana. Devido a essa ameaça constante referente ao mal funcionamento, é preciso encontrar formas de controlar o comportamento de tais dispositivos.

Outro fator de dificuldade ao controle de software de tais dispositivos é que, em geral, os dispositivos *smart*s podem ser reprogramados, seja para atualizações de suas funcionalidades, correção de *bug*s ou adição de funções ao dispositivo (DENG; HAN; MISHRA, 2006). A segurança da informação então passa a ser importante para assegurar que tais dispositivos não possam ser reprogramados com software maliciosos o que levaria a comportamentos indesejados do dispositivo e do sistema do qual faz parte. Dessa forma, é salutar dispor de uma forma de controle, tal qual a possibilidade de auditoria do software em execução no dispositivo. Essa forma de auditoria é fundamental em determinadas áreas de atuação, como a descrita na seção 1.1 a seguir.

1.1 MOTIVAÇÃO: METROLOGIA LEGAL

A metrologia é a ciência da medição e suas aplicações (JCGM 200:2012, 2012) e, é provavelmente a mais antiga ciência no mundo (HOWARTH et al., 2008), o que reflete sua importação para as relações humanas. A correta medição desempenha um papel fundamental nas ciências e nas inovações científicas e tecnológicas, bem como na qualidade na manufatura de produtos. Entre os benefícios da metrologia estão as justas trocas comerciais, o suporte a inovação e a proteção aos cidadãos (OIML, 2012). Com tantos benefícios ao mercado e a sociedade, há uma preocupação governamental em preservar a justa medição.

A metrologia legal é um ramo da metrologia que se preocupa com regulamentações legais para proteção de medições e instrumentos de medição originado a partir do desejo de preservar as justas transações comerciais (HOWARTH et al., 2008). Seu principal objetivo é assegurar aos cidadãos a correta medição quando da sua utilização para fins comerciais ou oficiais, embora também tenha papel na proteção do indivíduo e da sociedade como um todo, como por exemplo, através de regulamentação de medições utilizadas na área de saúde e de meio ambiente (OIML, 2012). Os governos devem prever proteções legais a instrumentos de medição, de forma a proteger mercados, consumidores e demais cidadãos. Entre os benefícios dessa prática estão: a redução de disputas jurídicas, a redução do custo de transação comercial, o controle de fraudes, entre outros (OIML, 2012). Para cumprir seu papel, o governo designa a Autoridade Metrológica, a qual é designada por lei para ser responsável pelas atividades de metrologia legal (INMETRO, 2016).

De forma a preservar a garantia metrológica, a Autoridade Metrológica deve especificar requisitos de desempenho para obrigatórios de sistemas de medição envolvidos em áreas consideradas importantes para um país, de acordo com suas políticas adotadas. O cumprimento a esses requisitos deve ser garantido através de medições e testes para checar a para que possa efetuar a garantia metrológica. Além disso, a Autoridade deve assegurar que os instrumentos mantenham as suas propriedades metrológicas durante as condições de uso e com o passar do tempo (OIML, 2012). A regulamentação também objetiva que o impacto das exigências sejam as mínimas possíveis, de forma que os benefícios de uma regulamentação ultrapasse suas interferências (Sinmetro; Conmetro; CBR, 2015). Em oposição à proteção, adversários maliciosos, interessados em modificar resultados de medição, podem agir no equipamento de forma a modificar sistematicamente resultados de medições para favorecer uma das partes envolvidas nas transações. A ação de tais adversários é facilitada principalmente em casa em que os instrumentos são implantados em ambientes hostis, sem supervisão constante, o que possibilita que possam

ter seus software modificados.

Paralelamente, instrumentos analógicos estão sendo paulatinamente substituídos por medidores com componentes eletrônicos com software embarcado. Nesses casos, o software é quem define, em última instância, o comportamento do instrumento, já que, através dele passam os dados provindos dos sensores que serão, por fim, processadas e exibidas ao usuário do sistema de medição. A complexidade adicionada com o software de medição aumenta a possibilidade de falhas (MACHADO et al., 2010).

Caso proteções de software não sejam adequadas, um adversário pode modificar o software de um equipamento de medição e disponibilizar *backdoor*s que podem ser explorados. Tais entradas escondidas podem ser utilizados de forma a modificar maliciosamente a informação de medição do instrumento, de instrumentos de medição de acordo com o interesse de uma das partes envolvidas na transação (BOCCARDO et al., 2010), o que caracteriza uma fraude metrológica. Um caso particularmente preocupante ocorre quando o comportamento malicioso presente pode ser acionado e desabilitado de acordo com o interesse do atacante, já que, em casos de fiscalização, nos quais tipicamente apenas testes de caixa preta são conduzidos, a fraude pode ser facilmente desabilitada pelo atacante (LEITãO; VASCONCELLOS; BRANDãO, 2014). O controle de software que está em execução na plataforma torna-se, então, essencial para proteção do correto funcionamento do sistema de medição.

A Autoridade Metrológica então passa a definir, para determinadas áreas e aplicações consideradas críticas, regulamentos e normas a serem cumpridos cumpridas pelos fabricantes de software para sistemas de medição. Tais regulamentos contém requisitos de cibersegurança de forma a proteger os instrumentos de medição contra mal funcionamento e reduzir vulnerabilidades passíveis de serem exploradas por atacantes. Um software a ser embarcado no sistema de medição é proposto para avaliação e testes de funcionalidade e testes de caixa branca são conduzidos. Por fim, apenas o software que é aprovado de acordo a regulamentação, pode ser utilizado no sistema de medição.

Adicionalmente à Avaliação de Modelo (INMETRO, 2016) descrita acima e de forma a prover a garantia metrológica, é preciso uma estrutura de fiscalização de forma a conferir se os instrumentos utilizados em campo são do mesmo modelo que foram previamente aprovados no processo de Avaliação de Modelo. Antes da adoção de software embarcados, inspeção visual do equipamento juntamente a testes de desempenho eram conduzidos na verificação ou perícia do instrumento; no entanto, devido ao seu caráter intangível, tais inspeções visuais não podem ser conduzidas no software. É necessário, por consequência, um mecanismo que possa comprovar que o software em execução em

um instrumento é uma cópia fiel do software que passou pelo processo de Avaliação de Modelo. Daí provêm a necessidade da exigência de um mecanismo chamado de verificação de integridade de software, o qual objetiva atestar que o software em execução é o mesmo que foi previamente aprovado por uma autoridade competente (PRADO et al., 2014).

1.2 OBJETIVOS DO TRABALHO

Devido à importância do cenário de proteção de software em metrologia legal e à necessidade de um método de auditoria de softwares presentes em instrumentos de medição no presente trabalho é proposto um método de verificação de integridade desenvolvido levando-se em consideração a segurança do método e a simplicidade de implementação para sistemas embarcados, especialmente em sistemas de medição.

O trabalho propõe um conjunto de algoritmos a serem implementados nas entidades participantes de um mecanismo de verificação de integridade. Adicionalmente aos algoritmos baseados em software mostrados da literatura, o presente método associando o resultado da verificação de integridade a uma característica do hardware, o número de ciclos de *clock* utilizados para o cálculo da resposta da verificação de integridade. Os algoritmos aqui propostos utilizam, além do conteúdo da memória de programa, referências dinâmicas de sua própria execução para gerar evidências do software em execução no instrumento. Ao contrário de outros trabalho presentes na literatura, a utilização de características da plataforma para a composição do resultado de verificação de integridade foi realizada sem a utilização de hardware adicional, o que permite a sua utilização em cenários com restrições de custos.

O protocolo proposto foi pensado para sistemas embarcados, já que sistemas de medição baseados em microcontroladores são cada vez mais utilizados. A Welmec (Welmec, 2011) classifica tais sistemas de medição como de *Tipo P*, em que todo o software de aplicação foi construído para a aplicação de medição (Welmec, 2012). Geralmente, tais sistemas embarcados são de baixo custo, o que justificaria a sua utilização em dispositivos de medição fabricados em grande escala. Do ponto de vista de segurança da informação, a ausência de um Sistema Operacional (SO), ou a utilização de SOs simples e sem acesso externo representa uma vantagem já que as vulnerabilidades de um SO não podem ser exploradas para a realização de ataques.

Após a definição dos algoritmos, apresenta-se a implementação para utilizando de uma ferramenta de prototipagem eletrônica e de um computador pessoal. Apresenta-se os resultados do esquema de verificações de integridade na presença de dois tipos de ataques. Por fim, compara-se também a segurança com um dos esquemas propostos na

literatura.

1.3 ORGANIZAÇÃO DO TRABALHO

No capítulo 2 é apresentado o que é a verificação de integridade e como o problema é tratado, no capítulo 3 são apresentados trabalhos de literatura que lidam com o problema de verificação de integridade. No capítulo 4 é proposto um novo mecanismo de verificação de integridade e no capítulo 5 são apresentados os resultados do esquema de verificação de integridade na presença de ataques. Por fim, no capítulo 6 é apresentada a conclusão do trabalho e trabalhos futuros.

.

2 INTEGRIDADE DE SOFTWARE

A preocupação com integridade de software tem se aumentado a medida em que dispositivos com software embarcado tem se tornado cada vez mais sofisticados e amplamente utilizados (SPINELLIS, 2000). A partir da verificação da integridade da memória de programa, pode-se deduzir que o software em execução na plataforma é o mesmo que foi programado de acordo com as especificações requeridas para o cumprimento de seus objetivos. Ao gerar evidências que apontam se um determinado software rodando em uma plataforma foi modificado ou não, a verificação de integridade de *software* pode ser utilizada para detectar possíveis alterações indevidas no programa em execução.

Estas alterações indevidas no software em execução são possíveis porque as **partes interessadas** não tem controle sob o ambiente em que o dispositivo opera. Conforme Spinellis (2000), **Partes interessadas**, ou *stakeholders* , são entidades que tem interesse na proteção da integridade de software, seja por interesses legais, regulatórios ou de negócio. Em tal ambiente não controlado, ou hostil, a alteração de software sob execução pode ocorrer de forma acidental ou intencional.

As alterações indevidas em software podem ser causadas de forma acidental por modificações na memória de programa ocorrida por falha de hardware (BEZ et al., 2003). Esse tipo de erro torna-se raro à medida que as técnicas de manufatura de equipamentos eletrônicos são aperfeiçoadas para o fornecimento de dispositivos cada vez mais integrados e confiáveis.

As modificações intencionais, em contrapartida, são de uma preocupação crescente. *Software*s com proteções precárias podem vir a ser alvo de atacantes e uma modificação ilegal (ou fraudulenta) de um software embarcado poderia, nesse caso, ser feita de forma simples, sem a necessidade de um hardware adicional, de forma remota e em grande escala. Além disso, em um cenário de ataque e sem as proteções adequadas a dispositivos reprogramáveis, o software malicioso pode se propagar pela rede de forma a comprometer a sua infraestrutura (DE; LIU; DAS, 2009).

Mesmo que o software contenha proteções, as plataformas podem ser implantadas em ambientes hostis, em que são passíveis de serem manipuladas por pessoas não autorizadas. Esse cenário é comum quando se trata de sistemas de redes de sensores sem fio (PADMAVATHI; SHANMUGAPRIYA et al., 2009). Dessa forma, é oportuno

dispor de mecanismos para verificar se um dispositivo com software embarcado está se comportando de acordo com seus requisitos oficiais e pré-definidos. Ou, em outras palavras, é necessário ter à disposição um mecanismo para avaliar a integridade de software de tal plataforma.

Eldefrawy et al. (2012) separa os mecanismos de garantia de integridade de plataforma em 4 grupos: (i) atestação baseada em hardware, (ii) estabelecimento de raiz de confiança dinâmica, (iii) outras técnicas baseadas em hardware e (iv) atestação baseada em software . Os três primeiros tipos utilizam algum tipo de hardware adicional de forma a auxiliar o processo de verificação de integridade. Esse hardware adicional é invocado de forma a calcular o estado atual da plataforma e gerar evidências sobre o seu estado. Um exemplo de utilização de hardware para a verificação de integridade é o mecanismo de *boot* seguro (ARBAUGH; FARBER; SMITH, 1997), em que a plataforma é checada no momento de sua inicialização. Um ponto negativo dessa abordagem é que o projeto da plataforma deve contemplar a utilização desse hardware de segurança e o seu custo pode ser inviável para cenários com restrição de recursos (ARMKNECHT et al., 2013).

Uma outra alternativa, a qual é discutida no presente trabalho, é a verificação de integridade baseada em software. Esta é possível através da *introspecção* (COLLBERG; NAGRA, 2010), também chamada de *reflexão* em alguns trabalhos (CARMO; MADRUGA; MACHADO, 2009; CARMO; MACHADO, 2009; SPINELLIS, 2000), em que o próprio software sob execução calcula o extrato de determinadas porções de seu conteúdo de forma a gerar evidências a respeito do software em execução. O próprio software da plataforma realiza, sob requisição, cálculos que dependem do conteúdo da memória de programa e retorna informação capaz evidenciar seu conteúdo. Pontos positivos decorrem da não-necessidade de hardware: projeto de plataforma mais simples, menor custo e possibilidade de utilização em sistemas legados (SESHADRI et al., 2004).

2.1 VERIFICAÇÃO DE INTEGRIDADE BASEADA EM SOFTWARE

A verificação de integridade baseada em software é um protocolo para estabelecimento de confiança entre duas entidades fisicamente distintas. Nesse protocolo, um sistema Verificador, (Verificador), checa a integridade de um outro dispositivo alvo ou Dispositivo sob verificação (Disv) (ARMKNECHT et al., 2013). Conforme Castelluccia et al. (2009), as técnicas de verificação de integridade de software são baseados em um protocolo de pergunta resposta em que o Verificador desafia o Disv a computar um extrato (*checksum*) de sua memória (fig. 1). A comunicação entre as duas entidades é feita de forma direta, isto é, sem pontos intermediários de comunicação, como roteadores

(FRANCILLON et al., 2014).

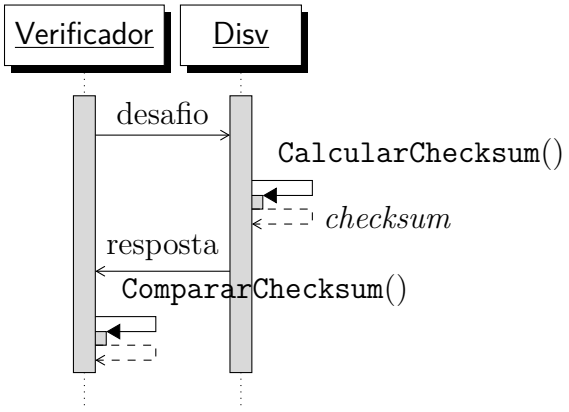

Figura 1 – Comunicação entre as entidades Verificador e Disv para a realização da verificação de integridade.

O Verificador é uma entidade confiável que está sob posse de uma autoridade fiscalizadora e não é submetida a ambientes não controlados ou hostis. No protocolo de verificação de integridade, o Verificador é responsável por requisitar as evidências do software em execução do Disv, bem como por conferir essas evidências recebidas com o valor esperado e apresentar o veredito. Conforme Seshadri et al. (2004), o Verificador é necessário já que, devido a ausência de um *hardware seguro*, não se pode confiar em uma auto-verificação de um Disv potencialmente malicioso.

O Disv é o dispositivo alvo da verificação de integridade de software. Considera-se que ele opera em ambientes não controlados, ou hostis. Uma vez em tal ambiente: (i) esse dispositivo pode vir a ser alvo de um ataque realizado por um adversário e, portanto, (ii) não se tem mais confiança de que software está em execução, logo, (iii) é um dispositivo não-confiável . Logo, o Disv, além de suas funcionais operacionais usuais, deve fornecer uma rotina de resposta a uma consulta de verificação de integridade, representada pela função `CalcularChecksum` na fig. 1. Tal rotina deve retornar um valor *checksum* dependente do estado interno do Disv, especialmente de sua memória de programa. Dessa forma, no protocolo de verificação de integridade, o Disv é responsável por gerar evidências do seu estado no momento, ou seja, do seu software em execução.

Ao final da verificação, o resultado *checksum* do cálculo é recebido pelo Verificador e ocorre um julgamento de acordo com tal valor recebido. Para conduzir tal decisão, o Verificador efetua uma comparação do valor recebido com o valor esperado para um Disv íntegro. Caso os dois resultados sejam similares, o software é considerado *Íntegro* caso contrário é considerado *Modificado*.

2.2 EXEMPLOS DE VERIFICAÇÃO DE INTEGRIDADE BASEADOS EM SOFTWARE

O esquema de verificação de integridade baseado em software apresentado anteriormente (fig. 1) é um modelo genérico. Alguns tipos de operações possíveis para efetuar a verificação de integridade de um Disv potencialmente malicioso são apresentados na presente seção.

2.2.1 *Dump* de memória

Uma primeira maneira é ***dump* de memória**, o qual é apresentado na fig. 2. Nesse tipo de verificação, o Verificador tem acesso direto à memória ou microcontrolador do Disv no qual o seu software está armazenado (PRADO et al., 2014). Com o devido acesso a essa memória, o Verificador deve enviar um comando específico (passo 1) e receber como resposta o conteúdo de cada um dos endereços da memória de programa do Disv (passo 2). O Verificador então compara o conteúdo recebido com o conteúdo esperado para um Disv íntegro (passo 3). Caso os conteúdos sejam iguais, o Disv é considerado íntegro, caso contrário, o Disv é considerado malicioso (passo 4).

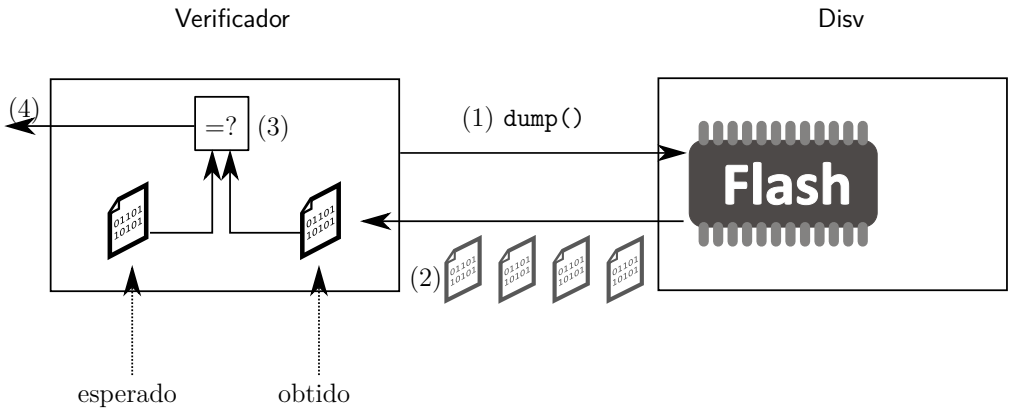

Figura 2 – Esquema de verificação baseado em *dump* de memória. O conteúdo extraído da memória flash é comparado com um valor esperado. Os números indicam a sequência das operações realizadas.

Um ponto positivo desse método é sua facilidade de implementação, visto que apenas é necessário retornar o valor armazenado na memória de programa. No entanto, a abordagem apresenta o revés de, em última análise, o software estar disponível para qualquer um com acesso ao dispositivo, o que é especialmente crítico para dispositivos implantados em ambientes hostis. Dessa forma, qualquer propriedade intelectual ou segredo industrial presente no software está comprometida. Além disso, conforme Prado et al. (2014), o processo de auditoria pode ser inconveniente, e o acesso direto à dispositivo onde está armazenada a memória de programa pode não ser prático ou possível.

2.2.2 Método baseado em resumo criptográfico

Nesse método, para efetuar a verificação de integridade, o Verificador solicita ao Disv no desafio, para que seja gerado um valor resumo, ou *checksum*, representativo de todo o conteúdo da memória de programa (fig. 3). Após o cálculo, o Disv envia o *checksum* ao Verificador, que então, compara o valor recebido com o valor esperado.

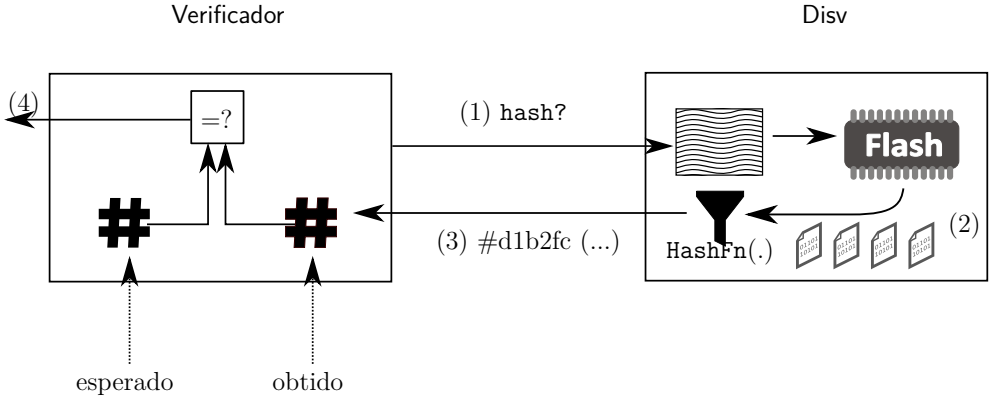

Figura 3 – Esquema de verificação baseado em resumo criptográfico. Um resumo do conteúdo de memória é enviada ao Verificador. Os números indicam a sequência das operações realizadas.

A função *hash* criptográfica, devido a sua propriedade de unidirecionalidade (STALLINGS, 2013), adéqua-se bem à tarefa de cálculo de *checksum*. O resumo gerado por tal função é representativo da sua entrada, no caso o conteúdo da memória de programa, e o valor retornado pelo Disv não permite a reconstrução do conteúdo completo da memória de programa do mesmo, favorecendo a confidencialidade. No entanto, um ponto negativo é que, para efetuar um ataque, um Disv malicioso pode simplesmente computar o resumo esperado em uma consulta e armazená-lo para retorná-lo durante uma consulta de verificação de integridade (seção 2.3).

2.2.3 Resumo criptográfico dependente de parâmetro

Um caso mais elaborado do *hash* completo de memória se dá com a adição, no desafio, de uma parâmetro aleatório $\mathcal{C}_i$. Geralmente é utilizado uma de uma semente aleatória s, ou *nonce*, a ser enviada do Verificador ao Disv (fig. 4). Tal semente é utilizada durante o cálculo do valor de resumo e este deve depender de seu valor. A utilização da semente deve garantir o "frescor" (*freshness*) do *checksum* resultado, ou, em outras palavras, o cálculo do valor de *checksum* somente pode ocorrer após o recebimento do desafio. Dessa forma, previne-se ataques de pré-computação, os quais serão apresentados na seção 2.3.

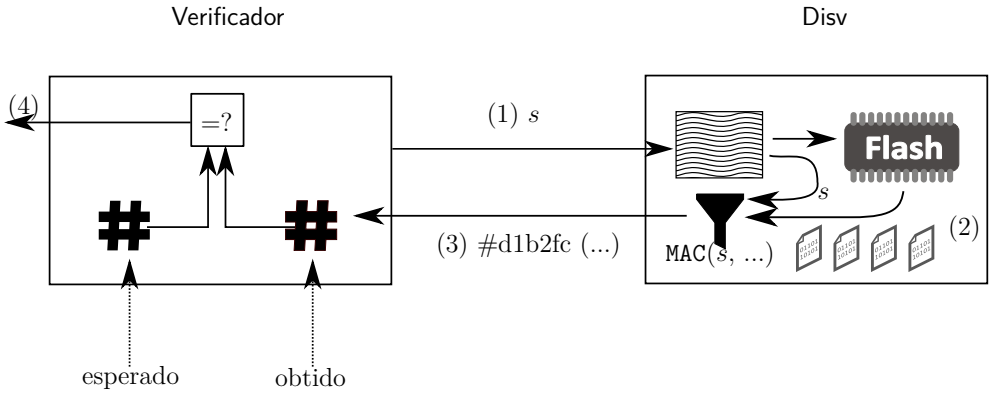

Figura 4 – Esquema de verificação baseado em resumo criptográfico com parâmetro. O resumo criptográfico não depende somente do conteúdo de memória, mas de uma semente s enviada junto ao desafio. Os números indicam a sequência das operações realizadas.

Em um exemplo de tal esquema, apresentado por Prado et al. (2014), é proposta a utilização de um algoritmo *Message Authentication Code* (MAC) para o cálculo do *checksum* a ser retornado ao Disv. O algoritmo MAC (STALLINGS, 2013), por sua vez, recebe dois parâmetros mostrados na eq. (2.1): um parâmetro de comprimento fixo K, ou chave; e um parâmetro de comprimento variável M, ou mensagem. Utilizando o conteúdo da memória de programa como o parâmero M, e um valor dependente da semente aleatória como a chave $K = f(s)$, pode-se garantir que somente é possível calcular o MAC após o recebimento do desafio com a semente s. Por padrão, pode-se considerar que $K = s$, exceção feita ao caso de o tamanho da semente ser maior que o comprimento permitido de K, neste caso deve-se utilizar uma função para mapear uma semente a um valor de K, como uma função *hash*, por exemplo. A função MAC, assim como a função *hash*, possui a propriedade da unidirecionalidade e, assim, a confidencialidade do código é mantida.

$$t = \texttt{MAC}(K, M) \tag{2.1}$$

2.2.3.1 Análise adicional de tempo de resposta

Buscando um maior nível de segurança do método de consulta com semente, é proposta a análise de tempo de resposta. Tal método de verificação de integridade é exemplificado por Seshadri et al. (2004) e descrito formalmente em Armknecht et al. (2013). Nesse método, o Verificador, ao enviar o desafio ao Disv, contabiliza o intervalo de tempo até o recebimento da resposta provinda do Verificador. Esse *tempo de resposta* é comparado com o tempo esperado de resposta para o caso de software íntegro e, caso não sejam similares, o Disv é considerado malicioso. Além da comparação dos resultados por

parte do Verificador, ocorre, portanto, uma análise de similaridade do tempo de execução do algoritmo de cálculo do *checksum*.

Uma de suas premissas é que o software íntegro deve implementar as rotinas relacionadas à verificação de integridade de forma ótima, ou seja, de forma mais eficiente possível com relação ao tempo de execução (SESHADRI; LUK; SHI, 2005; SESHADRI et al., 2004). A diferença do tempo de execução entre o software malicioso e o software íntegro deve-se ao fato de que um software malicioso, ao tentar emular o comportamento do software íntegro, irá responder com um certo atraso em relação ao software confiável. Um ponto negativo surge do fato de que demonstrar a otimização de implementação de um algoritmo complexo é um problema difícil (ARMKNECHT et al., 2013).

2.2.4 Resumo criptográfico sobre intervalos de memória

A técnica de intervalos de memória aleatórios, apresentada no trabalho de Spinellis (2000) e, mais tarde em Castro et al. (2013), utiliza, como parâmetro aleatório do desafio, a definição de um intervalo de memória em vez um único valor de semente (fig. 5). No desafio, o Verificador envia um par de valores, tal como (e_i, e_f), definindo um conjunto de endereços consecutivos da memória de programa do Disv; e em resposta o Disv retorna um extrato criptográfico representativo dos valores de cada um dos endereços de memória no intervalo correspondente. Ao contrário dos métodos considerados anteriormente, em que apenas uma consulta de verificação de integridade era necessária para obter uma valores representativos de toda a memória de programa do Disv, no presente método várias consultas devem ser conduzidas, cada uma delas questionando ao menos um endereço não listado nas pesquisas anteriores.

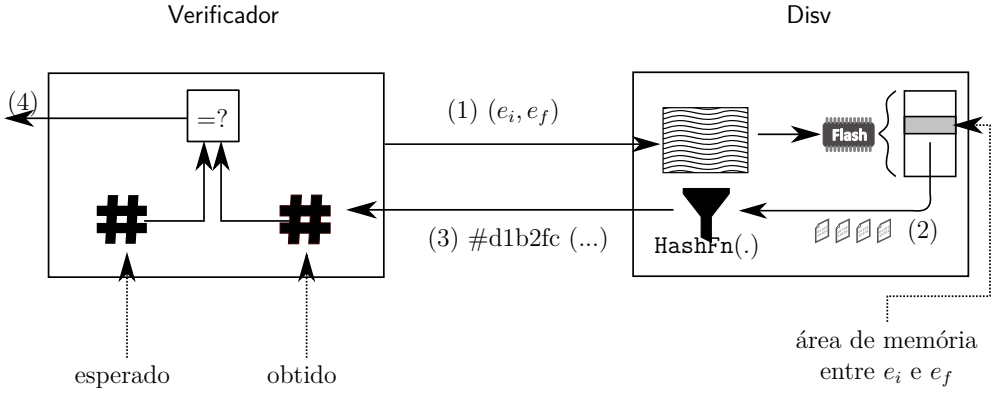

Figura 5 – Esquema de verificação baseado em intervalos de memória. O Disv retorna somente um resumo criptográfico da área informada pelo Verificador. Os números indicam a sequência das operações realizadas.

2.3 EXEMPLOS DE ATAQUES A ESQUEMAS DE VERIFICAÇÃO DE INTEGRIDADE

Um atacante, ou adversário, é uma entidade que tem interesse em modificar o software íntegro, **S**, em execução no Disv sem autorização dos *stakeholders* (SPINELLIS, 2000). Tal software inserido pelo atacante no Disv, ou *software malicioso*, $\tilde{\mathbf{S}}$, além de cumprir suas funções de acordo com as especificações do atacante, também deve ter como um de seus objetivos não ser discernível de um software íntegro em uma verificação de integridade. Contornando o mecanismo de verificação de integridade, o software malicioso pode manter a sua execução indetectável apesar de auditorias realizadas no Disv. Nessa seção, aborda-se algumas formas possíveis para o software malicioso contornar o esquema de verificação de integridade.

No **ataque de pré-computação**, o Disv malicioso mantém armazenado em memória o valor do *checksum* já esperado pelo Verificador (fig. 6). Após receber uma mensagem requisitando o cálculo da resposta de verificação de integridade, o software malicioso apenas opera uma consulta à memória de forma a obter o resultado esperado. O valor recuperado é então retornado ao Verificador.

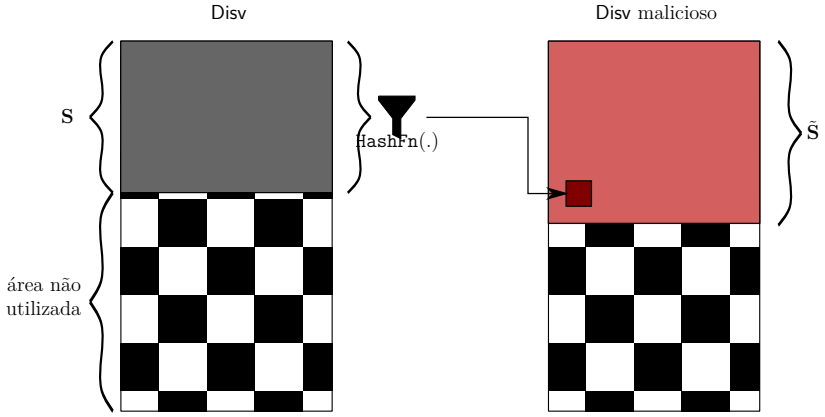

Figura 6 – Configuração da memória em um Disv íntegro e um Disv malicioso. O *checksum* do software íntegro é armazenado pré-computado em uma posição de memória do Disv malicioso

Da necessidade de evitar esse tipo simples de ataque, provém a necessidade de garantir o frescor do resultado retornado pelo Disv (seção 2.2.3). Em outras palavras, o esquema necessita garantir que o Disv somente possa calcular o valor de resposta após o recebimento da requisição de verificação de integridade pelo Verificador. Para cumprir essa tarefa, dentro do desafio, o Verificador envia também um desafio aleatório, $\mathcal{C}_i$, que é utilizado pelo Disv para calcular o resultado apropriado $\mathcal{R}_i$ (CASTELLUCCIA et al., 2009). Dessa forma, o Disv deve implementar uma função de cálculo tal que $\mathcal{C}_i \rightarrow \mathcal{R}_i$.

A quantidade de desafios possíveis $0 < i < 2^n = N$ deve também ser grande o suficiente, isto é, deve ser inviável para o Disv malicioso armazenar todos os pares

($\mathcal{C}_i \rightarrow \mathcal{R}_i$) de desafio e resposta possível e recuperá-los ao ser questionado em uma verificação de integridade. Uma quantidade de bits suficiente do desafio é utilizada para eliminar esse ataque, pois para um número suficientemente grande n de bits é necessário armazenar uma lista de tamanho 2^n de pares desafio resposta (PRADO et al., 2014). Salienta-se também que, para evitar a pré-computação de resultados parciais do cálculo, o valor de $\mathcal{C}_i$ deve ser utilizado desde o início do cálculo da resposta $\mathcal{R}_i$.

O **ataque de cópia de memória**, por sua vez, pode ser realizado com sucesso mesmo com a utilização do desafio $\mathcal{C}_i$. Para a construção do ataque, mantém-se uma cópia do software íntegro, **S**, em uma área não utilizada de memória de programa do Disv, $\mathscr{M}_{\mathcal{P}}$, a partir de um endereço a (fig. 7) (PRADO et al., 2014). Ao receber uma requisição de verificação, o Disv malicioso recupera os valores da memória a partir da área na qual está gravado o software íntegro, conforme a tab. 1. O ataque também explora o fato de que regiões não utilizadas de memória tem seus bytes preenchidos com um valor padrão D como `0x00` ou `0xFF`.

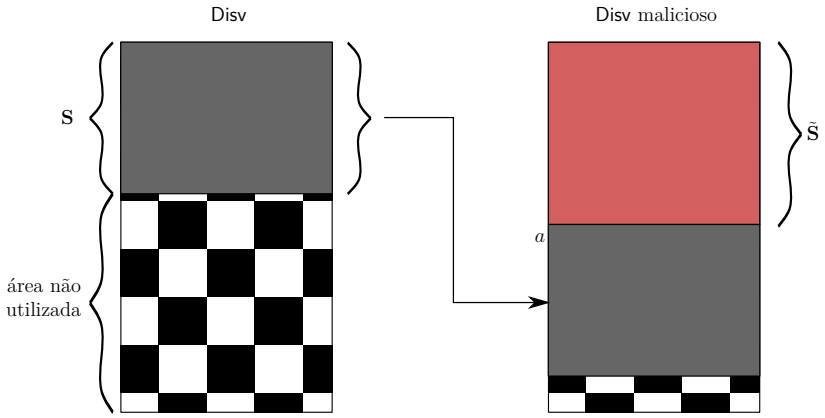

Figura 7 – Configuração da memória em um Disv íntegro e um Disv malicioso. Todo o software íntegro **S** é armazenado em uma área de memória não utilizada do Disv malicioso.

Tabela 1 – Comparativo das operações realizadas de consulta de memória entre **S** e $\tilde{\mathbf{S}}$. No caso de ataque, segunda linha da tabela, a recuperação dos valores de memória é realizado a partir da região em que está gravado **S**.

	Endereço consultado	Valor recuperado
S	m	$v \leftarrow \mathscr{M}_{\mathcal{P}}[m]$
$\tilde{\mathbf{S}}$	m	$v \leftarrow \begin{cases} \mathscr{M}_{\tilde{\mathcal{P}}}[a+m] & 0 \leq m \leq \texttt{sizeof}(\mathscr{M}_{\mathcal{P}}) \\ D & \text{caso contrário} \end{cases}$

Para a implementação do ataque, necessita-se de ao menos `sizeof`(**S**)[1] bytes

[1] Em que `sizeof` é uma função que retorna o tamanho de uma entidade em Bytes. Nesse trabalho, utiliza-se dessa nomenclatura como uma facilidade sintática referenciando-se indiferentemente ao tamanho de um software ou da memória de programa, por exemplo.

não utilizados para que possa-se armazenar o programa íntegro. Como alternativa, seria plausível armazenar somente as informações de áreas em de diferenças entre $\mathbf{S}$ e $\tilde{\mathbf{S}}$, de forma similar à saída do utilitário `diff` (IEEE; GROUP, 2016), dado que os dois podem compartilhar partes comuns do programa. A informação de diferenciação possivelmente requer menos memória disponível, no entanto, a operação de consulta a memória deverá possuir mais operações de desvios condicionais de forma a recuperar o valor que seria retornado por $\mathbf{S}$.

Como defesa para o referido ataque, propõe-se o preenchimento aleatório de áreas não utilizadas da memória de programa (SESHADRI et al., 2004). Dessa forma, não haveria o caso padrão de recuperação do valor D, conforme representado no caso $\tilde{\mathbf{S}}$ da tab. 1.

Outra variação do ataque de cópia de memória é o **ataque de compressão**. Esse ataque utiliza-se da compressão das informações de diferenciação entre $\mathbf{S}$ e $\tilde{\mathbf{S}}$ e o próprio $\mathbf{S}$ (SPINELLIS, 2000). Dessa forma, o atacante deve gerar espaço livre suficiente para armazenar as informações de diferenciação entre $\tilde{\mathbf{S}}$ e $\mathbf{S}$ e, assim, conduzir o ataque de cópia de memória. Nota-se também que é necessário embarcar uma rotina de descompactação (ou aproveitar uma rotina de descompactação já embarcada) de forma a recuperar os valores originais da memória de forma a conduzir a verificação de integridade.

A necessidade de descompactação e da consulta das diferenças entre $\mathbf{S}$ e $\tilde{\mathbf{S}}$ impacta o tempo de execução da rotina de cálculo. Apesar de os resultados retornados pelo Disv fraudulento serem corretos, o tempo de execução aumentado, caso seja analisado corretamente, pode acusar a fraude.

3 TRABALHOS RELACIONADOS

No presente capítulo apresenta-se alguns trabalhos recuperados da literatura relacionados à verificação de integridade de software. De forma a obter uma boa abrangência na cobertura da revisão bibliográfica, foi realizado um processo de pesquisa foi inspirado no modelo de revisão sistemática apresentado por Kitchenham (2004) e no modelo apresentado posteriormente por Biolchini et al. (2005). Tais estudos apresentam e exemplificam esquemas de Revisão Sistemática de Literatura.

Segundo Kitchenham et al. (2010), a Revisão Sistemática da Literatura é um meio de agregar conhecimentos sobre tópicos de engenharia de software ou sobre questões definidas de pesquisa. No presente trabalho está-se interessado na realização de "estudos de mapeamento", *mapping studies*, isto é, uma procura e classificação de estudos realizados sobre um determinado tópico específico da área. O processo de revisão bibliográfica foi conduzido nas seguintes etapas:

a) *Formulação de problema*, fase em que é definida a questão central de pesquisa e em que o pesquisador constrói definições que permitem distinguir entre o material relevante ou não-relevante;

b) *Coleta de dados*, fase em que são definidos os procedimentos para achar evidências relevantes do que foi definido no estágio anterior e também inclui a definição das fontes;

c) *Avaliação dos dados*, fase em que são aplicados os critérios de qualidade a separarem os estudos relevantes dos não relevantes e também são definidos os tipos de informações que devem ser extraídas dos estudos primários.

3.1 FORMULAÇÃO DO PROBLEMA

Com a presente revisão sistemática, pretende-se mapear o cenário das soluções de verificação de integridade, de forma a verificar questões como as proteções oferecidas por tais algoritmos e quais as dificuldades de sua implantação em cenários reais.

3.1.1 Questão de pesquisa

Segundo Kitchenham e Charters (2007), a definição das questões de pesquisa é a fase mais importante da pré-revisão e até mesmo um piloto de revisão de literatura pode ser conduzido para definir-se tais questões. As questões de pesquisa guiam o processo de

revisão da literatura. O objetivo final de todo o processo deve convergir para a resposta a essas questões.

No presente trabalho, buscou-se determinar o estado da arte dos mecanismos de verificações de integridade para sistemas embarcados. Para isso é realizado um levantamento a respeito das técnicas de segurança especialmente àquelas que visam evitar que código malicioso execute em tais sistemas, ou então na detecção de que tais códigos ilegais estejam em execução no momento da verificação. A questão de pesquisa sumariza esse objetivo:

> *Quais técnicas de verificação de integridade baseadas em software são aplicáveis a sistemas embarcados?*

3.1.2 Grupos de termos utilizados

A partir da questão de pesquisa, grupos de termos foram destacados os seguintes termos para a posterior montagem das *string*s de pesquisa: *software*, *verificação*, *embarcado*, *baseado em software* e *integridade*. Utilizando esses termos como base foi conduzida uma consulta preliminar utilizando partes dos termos em questão e foram colhidos termos similares para compor a *string* de pesquisa.

Para capturar os termos chaves, utilizou-se de um *script*[1] para recuperar palavras-chave de um conjunto de artigos recuperados de uma pesquisa piloto prévia a respeito do assunto. As palavras-chave foram agrupadas e ordenadas pelo número de ocorrências, e os termos com duas ou mais ocorrências foram organizados e analisados. Ao final, palavras-chave relevantes foram adicionadas aos grupos de pesquisa pertinentes, e os termos resultantes são mostrados na tab. 2, na qual cada ID corresponde a um grupo, o qual contém termos sinônimos ou de significados similares para o contexto em questão.

Tabela 2 – Grupos de termos similares utilizados como chaves nas strings de busca. Os IDs de cada grupo representam o grupo em questão na montagem da *string* de pesquisa.

ID	Grupos de termos similares
software	“software”, “firmware”, “application”, “code”
attestation	“attestation”, “verification”
embedded	“embedded”
modes	“software-based”, “introspection”, “self checksumming”
integrity	“integrity”

[1] disponível em <https://gitlab.com/crisgc1/search-str-builder.git>

3.1.3 Strings de pesquisa

Para a formação das strings de pesquisa, são utilizados os operadores lógicos de disjunção ($\vee$) e o o operador lógico de conjunção ($\wedge$). As strings de pesquisa são as seguintes:

a) SP1: *modes* $\wedge$ *integrity* $\wedge$ *attestation* $\wedge$ *software*

b) SP2: *integrity* $\wedge$ *attestation* $\wedge$ *embedded* $\wedge$ *software*

Os termos enfatizados correspondem a um ID de grupo (tab. 2) e para a construção da string de pesquisa são expandidos para todos os termos do respectivo unidos através de conectores $\vee$. Levando em consideração também as limitações no tamanho das *string*s definidas pelas bases de pesquisa, definidas na seção 3.2, optou-se por dividir os termos em duas expressões distintas.

3.2 COLETA DE DADOS

A busca dos materiais para a pesquisa foram efetuadas por bases de dados digitais *online*, devido a praticidade de busca por palavras chaves oferecida em tais plataformas, bem como pelo amplo acervo oferecido. Segundo sugestão de pesquisadores mais experientes na área, as fontes de pesquisa pesquisadas foram as bases de dados do IEEE (IEEE, 2017) e da ACM (ACM, 2017). Para analisar os artigos recuperados a partir dessas bases de pesquisa, utilizou-se de metodologia que compreende os seguintes passos:

a) leitura de título;

b) leitura do abstract;

c) leitura da introdução;

d) leitura da conclusão;

e) leitura do artigo completo.

3.3 AVALIAÇÃO DE DADOS

Para a filtragem dos artigos pré-selecionados pela pesquisa, foram definidos critérios de inclusão/exclusão. Os critérios de inclusão englobam os seguintes pontos:

a) os estudos avaliados são escritos em língua inglesa;

b) a publicação deve ser recente, isto é a partir de 2011;

c) apresenta uma descrição do mecanismos de verificação;

d) apresenta a descrição dos experimentos realizados;

e) os experimentos são reprodutíveis.

3.4 EXECUÇÃO DA PESQUISA

Para automatizar a geração de *string*s de pesquisa para as plataformas escolhidas, foi criado um *script* [2] que recebe como entrada um documento YAML (EVANS, 2017) com as definição dos grupos de termos e também de uma estrutura de árvore representando cada uma das *string*s de pesquisa. A partir dessa definição, as seguintes pesquisas foram lançadas em cada um das plataformas de pesquisa escolhidas (seção 3.2):

a) String de pesquisa SP1
   - IEEE: (("software-based"OR introspection OR "self checksumming") AND (integrity) AND (attestation OR verification) AND (software OR firmware OR application OR code));
   - ACM: (("software-based"or introspection or "self checksumming") and (integrity) and (attestation or verification) and (software or firmware or application or code))

b) String de pesquisa SP2
   - IEEE: ((integrity) AND (attestation OR verification) AND (embedded) AND (software OR firmware OR application OR code));
   - ACM: ((integrity) and (attestation or verification) and (embedded) and (software or firmware or application or code)).

Tabela 3 – Quantidade de artigos retornados. Os valores foram separados pelas *string*s de pesquisa e pelas bases de pesquisa.

	IEEE	ACM
SP1	9	234
SP2	50	892

As quantidades de artigos retornados em cada uma das consultas realizadas são mostrados na tab. 3. Após passar pelas etapas de filtragem, tivemos as seguintes quantidades de arquivos recuperados:

a) Após a análise do *abstract*: 58 entradas relevantes

[2] disponível em https://gitlab.com/crisgc1/search-str-builder.git

b) Após a análise da introdução: 23 entradas relevantes

c) Após a análise da conclusão: 12 entradas relevantes

Outros trabalhos relevantes descobertos fora do contexto da revisão bibliográfica também foram levados em considerações devido à sua relevância e também serão tratados. Como ferramenta para auxílio na tarefa de organização dos artigos utilizando a ferramenta JabRef (JABREF, 2017).

3.5 RESULTADOS

Como resultado do processo de revisão bibliográfica, diversos trabalhos relevantes foram encontrados, os quais são apresentados e discutidos brevemente nesta seção.

Armknecht et al. (2013) apresenta uma base formal para métodos de verificação de integridade e estudos de seus requisitos de segurança. No trabalho também é apresentado um esquema genérico de verificação de integridade e uma análise da segurança deste através da formalização. O trabalho supõe que o algoritmo de cálculo de *checksum* tem implementação ótima com relação ao tempo de execução. Alguns conceitos, como a definição das entidades e aspectos das fases de verificação de integridade são baseados no trabalho acima.

Francillon et al. (2014) aborda um problema similar ao da verificação de integridade: a *atestação remota*, o qual adiciona um nível de dificuldade ao considerar que o Verificador e o Disv não se comunicam diretamente, mas através de uma rede possivelmente com elementos intermediários. No trabalho, os autores afirmam que identificaram propriedades suficientes e necessárias para para um dispositivo implementar a atestação remota de forma segura.

Um dos trabalhos pioneiros a abordar o problema de verificação de integridade puramente baseados em software é apresentado por Spinellis (2000). Nele, propõe-se o mecanismo da introspeção, ou reflexão, de forma a permitir a verificação de integridade baseando-se somente em software e não na presença de um hardware confiável. Também introduziu-se a ideia de um desafio aleatório, representado pelo desafio baseado em regiões aleatórias de memória, conforme apresentado na seção 2.2.4.

No trabalho de Seshadri et al. (2004) é apresentado uma implementação de verificação de integridade para dispositivos embarcados, baseando-se somente em protocolo desafio-resposta chamado *Swatt*. O Swatt define um algoritmo de cálculo de *checksum* a ser implementado em um Disv de forma a gerar um valor representativo

de toda a memória de programa (resposta) a partir de uma semente (desafio). Como desafio, o Swatt recebe um número aleatório o qual irá definir um percurso de acessos a memória de programa de modo a atualizar o valor do checksum. Como forma de definir se o software em execução no Disv é íntegro ou não, o classificador analisa, além do valor de *checksum* retornado como resposta, o tempo de execução do algoritmo. Um ponto negativo decorre da necessidade de implementar de forma ótima o algoritmo de *checksum*. O presente trabalho utiliza-se uma ideia similar, no entanto, adicionamos mais uma dificuldade à tarefa de emulação do comportamento do Disv íntegro pelo atacante

Em um trabalho posterior, Seshadri, Luk e Shi (2005) apresentam um passo adicional ao abordar o estabelecimento de uma raiz de confiança dinâmica, isto é, um pedaço de software confiável, em PCs tradicionais, de forma a permitir um esquema de verificação de integridade também por uma abordagem baseada em software. O trabalho de Li, McCune e Perrig (2011) apresentam um mecanismo de verificação de integridade para periféricos de computador, tal artigo exige também a necessidade de uma implementação simples e otimizada do algoritmo de *checksum*.

Um trabalho posterior Carmo e Machado (2009) baseia-se no trabalho anterior de Seshadri et al. (2004) para também proporem um método de verificação de integridade baseado em software. Na proposta, os autores enviam uma lista de endereços da memória de programa que desejam consultar como desafio a ser enviado ao Disv, em uma abordagem semelhante à da seção 2.2.4, exceto que os endereços são definidos um-a-um. Também é proposto a utilização de algoritmos de resumo criptográfico (*hash*), e não apenas um algoritmo de *checksum* elaborado de forma simples.

Nos trabalhos de Horsch et al. (2014) e Zhao et al. (2014) são apresentadas uma forma de estabelecer uma raiz-de-confiança em processadores ARM. Tal raiz de confiança é um setor reconhecidamente íntegro no dispositivo sob análise. A solução de Horsch et al. (2014) também se baseia na otimalidade de implementação do código de verificação de integridade, apresentando inclusive uma construção cuidadosa das funções de geração de números pseudo-aleatório e cálculo de *checksum*. Zhao et al. (2014) utiliza a característica de *TrustZone* (Arm Limited, 2017) o que deixa a sua implementação dependente dessa tecnologia e de seu fabricante.

Ismail, Syed e Musa (2014) discutem o monitoramento do comportamento da aplicação com relação a chamadas de sistema, utilizando uma abordagem de atestação comportamental. Porém seu método pode não ser aplicável para sistemas embarcados, visto que não há sistema operacional. Além disso, há a utilização de TPM, o que torna a abordagem dependente de hardware adicional.

Os trabalhos de Kocabas et al. (2011), Kong et al. (2014) e Schulz, Sadeghi e Wachsmann (2011) propõem a utilização de *Physically Unclonable Functions* (PUFs) (MAES, 2013) de forma a integrar o mecanismo de verificação de integridade entre hardware e software. Apesar de mais simples que um TPM, o PUF ainda representa um hardware adicional a ser incluído na plataforma, o que pode ser impraticável em cenários de baixo custo.

Kovah et al. (2012) apresenta um esquema de atestação remota baseado em tempo para computadores pessoais em ambientes corporativos. Os autores alegam que, mesmo com o atraso provido pela rede, conseguem bons resultados. Entretanto, o esquema também é baseado em hardware, já que utiliza funcionalidades de TPM (ISO.ORG, 2016). O trabalho também admite que a sua implementação do algoritmo da rotina de extrato (*checksum*) pode não ser ótima. Similarmente, Rauter et al. (2015) apresenta um mecanismo de atestação remota baseado em recursos. Neste artigo, há a utilização de hardware adicional.

Uma abordagem semelhante é apresentada por Preschern et al. (2013), a qual propõe um mecanismo de autenticação para ambientes de automação, mais especificamente sistemas supervisórios. A abordagem citada nesse artigo utiliza criptografia de caixa branca, em que um de seus objetivos é que, mesmo o atacante tendo acesso ao equipamento a ter a integridade verificada (Disv), ele não poderá recuperar uma chave secreta. Esse abordagem difere da do presente trabalho visto que, aqui, considera-se que dado tempo suficiente a um atacante, este descobrirá qualquer segredo escondido no código fonte.

Os trabalhos de Srinivasan, Dasgupta e Gohad (2011) e Yan et al. (2011) também apresentam um mecanismo de verificação de integridade para PCs convencionais. O primeiro apresenta um mecanismo de atestação que utiliza injeção de código de forma a estabelecer uma raiz de confiança na entidade sob análise. Além de necessitar de uma entidade confiável para que forneça o código a ser injetado para a plataforma verificadora, a sua utilização fica prejudicada em plataformas que não aceitam a injeção de código em tempo de execução. Yan et al. (2011) foca em plataformas multicore, mas ainda depende da propriedade de otimalidade de código que executa o cálculo de *checksum*.

Alguns trabalhos como Ambrosin et al. (2016) e Asokan et al. (2015) apresentam mecanismos de verificação de integridade para o ambiente de redes de sensores, possivelmente com um grande número de nós. Utilizam arquiteturas propostas na literatura, tais como SMART (ELDEFRAWY et al., 2012), de forma a efetuar a verificação entre os nós da rede. Dessa forma, podem estar sujeitos à necessidade de componentes

de hardware adicionais para a execução da verificação.

Sorber et al. (2012) apresentam mecanismos de verificação da integridade de informação para sensores utilizados na área médica para monitoramento de pacientes. A abordagem citada necessita de equipamentos de hardware adicionais como *smart card*s e *smart phone*s o que torna difícil sua adoção em cenários de custos limitado.

Por fim, Castelluccia et al. (2009) apresenta um estudo da viabilidade de esquemas puramente baseados em softwares, abordando suas fraquezas, especialmente em métodos que se baseiam na implementação otimizada do algoritmo de checksum, ou algoritmo de cálculo.

3.6 ANÁLISE DOS RESULTADOS DA PESQUISA

De forma a organizar o resultados da pesquisa separou-se os resultados de acordo com o assunto dos artigos conforme a tabela 4. Divide-se os artigos nas seguintes categorias:

a) *Base e discussões*, com estudos teóricos a respeito dos mecanismos de verificação de integridade, bem como discussões a respeito da sua viabilidade;
b) *Verificação de integridade*, com esquemas de verificação de integridade baseados em software;
c) *Atestação remota*, com estudos de verificação de integridade através de redes de comunicação;
d) *Hardware adicional*, com mecanismos de verificação de integridade baseados em hardware;
e) *Injeção de código*, com esquemas de verificação de integridade em que ocorre injeção de código.

O presente trabalho utilizou os trabalhos categorizados como *Base e discussões*, como modelos na confecção da proposta apresentada na seção 4. Em especial, o trabalho de Armknecht et al. (2013) foi utilizado como fonte para a definição das entidades bem como os nomes das fases que compõem os mecanismos de verificação de integridade. O presente trabalho incrementa os mecanismos de verificação de integridade tais como (SESHADRI et al., 2004), adotando, no cálculo do *checksum*, outras características específicas dependente da plataforma, sem a necessidade de adição de hardware especial, o que representa uma vantagem aos trabalhos como (ISMAIL; SYED; MUSA, 2014). Finalmente os trabalhos de *atestação remota* focam no problema de verificação de

Tabela 4 – Categorização dos artigos

Características	Trabalhos relacionados
Base e discussões	(ARMKNECHT et al., 2013), (FRANCILLON et al., 2014), (CASTELLUCCIA et al., 2009)
Verificação de integridade	(SPINELLIS, 2000), (SESHADRI et al., 2004), (SESHADRI; LUK; SHI, 2005), (LI; MCCUNE; PERRIG, 2011), (CARMO; MACHADO, 2009), (HORSCH et al., 2014), (YAN et al., 2011)
Atestação remota	(FRANCILLON et al., 2014), (AMBROSIN et al., 2016), (ASOKAN et al., 2015)
Hardware adicional	(ZHAO et al., 2014), (ISMAIL; SYED; MUSA, 2014), (KOCABAS et al., 2011), (KONG et al., 2014), (SCHULZ; SADEGHI; WACHSMANN, 2011) (PRESCHERN et al., 2013)
Injeção de código	(SRINIVASAN; DASGUPTA; GOHAD, 2011)

integridade através de redes com múltiplos nós intermediários, e utilizam como base, outros esquemas de verificação de integridade, podendo requerer também modificações no projeto do hardware, como Asokan et al. (2015).

4 EVINCED: ESQUEMA DE VERIFICAÇÃO

O presente trabalho propõe EVINCED, um novo esquema de verificação de integridade, composto de algoritmos e um protocolo de comunicação a ser implementado pelo Verificador e pelo Disv. É proposta uma metodologia de verificação de integridade que visa adicionar um grau de dificuldade a um atacante emular o comportamento de um Disv não-íntegro.

A proposta apresentada aqui utiliza a contabilização de ciclos de execução do algoritmo de cálculo de *checksum* de forma a analisar o resultado do esquema de verificação de integridade. Um ponto positivo da contabilização dos ciclos de execução, é que, ao contrário do tempo de execução do algoritmo, eles não são afetados por fatores como a latência de comunicação ou precisão do relógio de tempo real utilizado. Por outro lado, como o número de ciclos é retornado pelo Disv, um Disv malicioso, isto é, modificado por um atacante, poderia, antes de responder a uma requisição de verificação de integridade, modificar ou gerar um resultado de forma a emular o comportamento de um Disv íntegro. Ainda assim, a necessidade de calcular os comportamento de um Disv íntegro retornando os seus ciclos de *clock* representam uma dificuldade adicional ao atacante.

Assim sendo, é proposto um novo esquema de verificação de integridade que é baseado somente em software, e implementação simples. Como apresentado no capítulo 2, no presente trabalho separa-se o esquema de verificação de integridade em 3 estágios ou etapas, respectivamente: Desafio, Cálculo e Classificação.

4.1 PREMISSAS E MODELOS DE ATAQUE PARA O EVINCED

De forma a entender e delimitar o cenário com que está-se lidando no presente trabalho, foram assumidas certas condições. Nessa seção discute-se o modelo de ataque bem como outras premissas para a proposta.

4.1.1 Objetivo do atacante e Fases do ataque

O atacante objetiva modificar o software embarcado no Disv, e, portanto, modificar o comportamento do dispositivo de forma a servir aos seus propósitos. O atacante intenciona ainda que tal modificação não possa ser detectada por uma consulta de verificação de integridade realizada no dispositivo modificado. Dessa forma, o atacante deve programar o software malicioso $\tilde{\mathbf{S}}$ de maneira que responda corretamente a um

desafio de verificação de integridade lançado pelo Verificador, ou seja, de forma similar ao comportamento esperado para um software íntegro.

O modelo de ataque descrito no presente é baseado no proposto em (ARMKNECHT et al., 2013), no qual divide-se o ataque em duas fases distintas: fase de preparação e fase de execução. Na fase de preparação um atacante consegue um exemplar de Disv e extrai o programa íntegro $\mathbf{S}$ através da utilização de técnicas de engenharia reversa. O atacante, tendo posse do código, tem acesso a todas as chaves e demais segredos presentes no código. Após efetuar a engenharia reversa, o atacante pode inserir um código malicioso $\tilde{\mathbf{S}}$ em Disv, o qual possui um comportamento não condizente com o esperado. No presente modelo de ataque, não há limite de tempo ou custos imposto ao atacante durante a fase de preparação.

Na fase de execução, o Disv modificado com código malicioso deve responder corretamente a uma requisição de verificação de integridade. O atacante deve preparar $\tilde{\mathbf{S}}$ de tal forma que a resposta recebida pelo Disv seja similar ao comportamento esperado para $\mathbf{S}$.

4.1.2 Premissas

Apenas um microcontrolador/software. No presente trabalho, considera-se que o Disv é um dispositivo o qual possui apenas um microcontrolador embarcado, e tal microcontrolador permite apenas um software em execução. O cenário de um microcontrolador com somente um software é comum especialmente em cenários de baixo custo. Em todo o caso, se em um cenário de mundo real ocorrer a utilização de vários microcontroladores em um mesmo equipamento, pode-se implementar a rotina de verificação de integridade em cada um deles.

Instruções adicionais da plataforma. No presente trabalho, considera-se que a plataforma é capaz de ler seu próprio conteúdo da memória de programa. Isto é, a plataforma é responsável pode acessar/ler a parte da memória em que está gravado o código executável. Isso é necessário para que se possa utilizar da introspeção, ou reflexão, necessária para o cálculo do *checksum* (CARMO; MACHADO, 2009).

A plataforma de execução também deve ser capaz de prover uma instrução para acesso ao número de ciclos de *clock*. No presente trabalho utiliza-se de uma instrução chamada `CycleCount` de forma a poder contabilizar o número de ciclos de processamento. Tal instrução retorna o número de ciclos de *clock* desde a inicialização da plataforma, de forma similar à instrução `RDTSC` presente em processadores de arquitetura x86 (Intel Corporation, 2016), ou através do acesso ao registrador `PCCNT` em uma arquitetura

AVR32 (Atmel, 2011).

Engenharia reversa. Assume-se que um atacante pode obter uma cópia de um Disv íntegro e executar a engenharia reversa da mesma, de forma a obter e analisar o software embarcado em tal plataforma. Como foi visto, o dispositivo embarcado pode ser colocar em um ambiente hostil e sujeito a furtos, portanto a engenharia reversa do dispositivo é plausível.

Modificação de Hardware. Em contraste ao software, ao atacante não é permitido modificar qualquer componente de hardware do equipamento. Assume-se que a modificação de hardware pode ser facilmente detectada através de uma simples inspeção visual. Como um dos objetivos do atacante é evitar a detecção, ele não modificaria o hardware. A modificação do hardware pode ser evidenciada pela utilização de mecanismos de lacração ou de selagem, por exemplo.

Reprogramação do Disv. Também considera-se que o atacante pode modificar, com sucesso, o software embarcado no Disv. Como a modificação do software não irá incorrer em diferenças visíveis, o atacante pode reprogramar o microcontrolador à vontade.

Canal de comunicação. O canal de comunicação estabelecido entre os dispositivos de verificação de integridade fornece integridade e autenticação de dados. Também assume-se que a comunicação é estabelecida diretamente entre os dispositivos, ou, em outras palavras, a rede entre os dispositivos não tem dispositivos intermediários, como roteadores. Em resumo, quando um dispositivo recebe com sucesso uma mensagem do protocolo de verificação de integridade, admite-se que a mensagem veio de sua contrapartida e a mensagem não foi modificada intencionalmente.

4.2 FASES DO EVINCED

O protocolo de verificação de integridade define a interação entre as duas entidades – Verificador e Disv – e a presente seção apresenta as três fases, ou etapas, do esquema de verificação de integridade: Desafio, Cálculo e Classificação (fig. 8). As fases Desafio e Classificação são executadas pelo Verificador, enquanto o Cálculo é realizado pelo Disv. Na fase de Desafio, ocorre a definição de um desafio $\mathcal{C}$ a ser utilizado pelo Disv no cálculo de uma resposta $\mathcal{R}$ representativa do valor de seu estado. Esta etapa que calcula $\mathcal{R}$ a partir de $\mathcal{C}$ é chamada Cálculo. Por fim, o valor resultante da etapa de cálculo, $\mathcal{R}$, é analisada pelo Verificador de forma a definir se a resposta recebida é um resultado válido, no caso de Disv íntegro, ou inválido, no caso de Disv malicioso. Essa decisão de resultado final ocorre na etapa de Classificação.

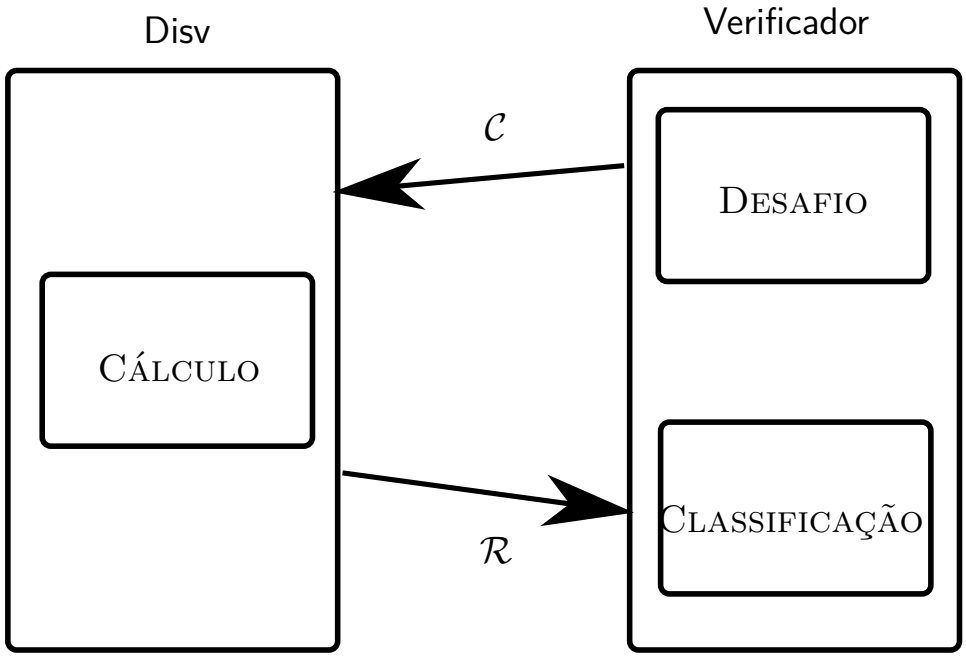

Figura 8 – Fases do EVINCED

Primeiramente, antes de poder enviar um desafio de verificação de integridade, deve-se estabelecer o canal de comunicação entre o Disv e o Verificador. Conforme mencionado na seção 4.1.2, o canal não está sujeito a ataques contra a integridade e autenticidade da mensagem. Uma vez estabelecido o enlace de comunicação, o esquema de verificação pode ter início.

Nas próximas seções apresentam-se os estágios do esquema de verificação de integridade. O estágio de Cálculo é considerado o coração da verificação de integridade, o algoritmo implementado para este estágio define o comportamento do esquema de verificação de integridade: o estágio Desafio é responsável por gerar suas entradas e o estágio Classificação é responsável por analisar suas saídas. Primeiramente apresentamos a etapa de Cálculo.

4.2.1 Cálculo

A fase de cálculo constitui a principal fase da verificação de integridade sendo responsável por gerar a evidência do código em execução no Disv. É realizado pelo Disv após o recebimento de desafio $\mathcal{C}$ emitido pelo Verificador. Nessa fase, o Disv calcula um valor resultado $\mathcal{R}$, o qual depende do desafio $\mathcal{C}$ recebido. O pseudo-algoritmo da rotina de Cálculo, CalcAlg, é apresentado no alg. 1. Pode-se dividir a tarefas em três partes: Inicialização, Laço de cálculo e Finalização.

Entradas. Três valores são recebidos como parte do desafio: uma semente s,

Algoritmo 1 Algoritmo da fase de CÁLCULO: CalcAlg

Entrada(s): 1. Um desafio $\mathcal{C} = (s, p, bs)$; 2. O tamanho da memória de programa do Disv ms.

Saída(s): Uma resposta $\mathcal{R}$ calculada de acordo com o desafio $\mathcal{C}$ recebido.

1: $c_0 \leftarrow$ `CycleCount`()
2: $h \leftarrow$ `HashFn`(s)
3: `Prng.Seed`(s)
4: **enquanto** ¬`IsPrefix`(p, h) **faça**
5: $bl \leftarrow$ `Prng.Get`(`max` $= ms/bs$)
6: $v \leftarrow \mathscr{M}[(bl \times bs)...[(bl + 1) \times bs]]$
7: $h \leftarrow$ `HashFn`($h \| v$)
8: **fim enquanto**
9: $c_f \leftarrow$ `CycleCount`()
10: $c \leftarrow c_f - c_0$
11: $\mathcal{R} = (h,\, c)$

um critério de parada p e um tamanho dos blocos bs. Tais parâmetros gerados na etapa de DESAFIO definem e modificam o cálculo de uma forma diferente a ser indicada quando da explicação do algoritmo.

Saída. O algoritmo tem por finalidade calcular o valor de *checksum* – ou, nesse caso, de *hash* – h que depende do desafio recebido $\mathcal{C}$. O valor h é representativo do conteúdo da memória de programa, e, portanto, é representativo do software em execução na plataforma do Disv. Junto ao valor h também é retornado um valor c indicativo da quantidade de ciclos de processamento (*clock*) que foram utilizados em todo o processo de cálculo. O resultado final $\mathcal{R}$ corresponde, então, um par ordenado contendo, respectivamente, o valor de hash e número de ciclos: $\mathcal{R} = (h, c)$.

Inicialização. Em primeiro lugar, o Disv inicializa o valor de *hash* h e um Gerador de Números Pseudoaleatório ou simplesmente PRNG, na sua sigla em inglês (ll. 2 e 3 do alg. 1). O valor inicial de h é definido como uma função de *hash* aplicada sobre a semente recebida s, parte do desafio $\mathcal{C}$. O estado inicial do PRNG, também é inicializado com a semente s. O PRNG é utilizado posteriormente no laço de cálculo para definir um caminho aleatório de endereços de memória a serem visitados de forma a atualizar o valor de h. Depois desses passos, a inicialização do algoritmo é concluída e as rodadas de cálculo para atualização do valor de h podem iniciar.

Laço de Cálculo. O laço de cálculo é responsável pela computação do valor de hash h, o qual é parte da saída $\mathcal{R}$ do CalcAlg. O cálculo de h é feito iterativamente em várias "Rodadas". Em cada uma dessas rodadas de cálculo, o algoritmo executa três operações, respectivamente, (i) a seleção pseudo aleatório de um bloco de endereços

contíguos da memória de programa do Disv; (ii) a recuperação do valor de memória gravado em tais endereços contíguos; e, finalmente, (iii) a atualização do valor de hash com base nos valores recuperados da memória. Estes passos são representados nas ll. 5 a 7 do alg. 1.

Laço de cálculo – Divisão de memória em blocos. Para a implementação do algoritmo, considera-se a memória de programa do Disv como dividida em blocos de memória. Tais blocos nada mais são que a aglutinação de uma ou mais palavras de memória[1]. O tamanho dos blocos é um parâmetro de entrada do algoritmo, bs, e define a quantidade de bytes de memória que é recuperada em uma consulta da memória a cada rodada de cálculo. Os valores recuperados são utilizados na posterior atualização do valor de *hash* representativo dessa memória.

A divisão visa agilizar a execução do algoritmo, conforme será visto posteriormente na seção 4.2.2. De forma simplificada, a separação da memória em blocos com possivelmente mais de um byte, faz com que sejam necessárias menos rodadas de forma a consultar todas as posições de memória.

A fig. 9 representa a divisão de memória em blocos. Duas características do blocos são mostradas na figura, a primeira sendo que todos os blocos tem igual tamanho, a segunda, é que cada palavra (ou byte) da memória pertence a um único bloco. Com essas condições, o tamanho bs deve ser escolhido de forma a ser um divisor do tamanho da memória ms.

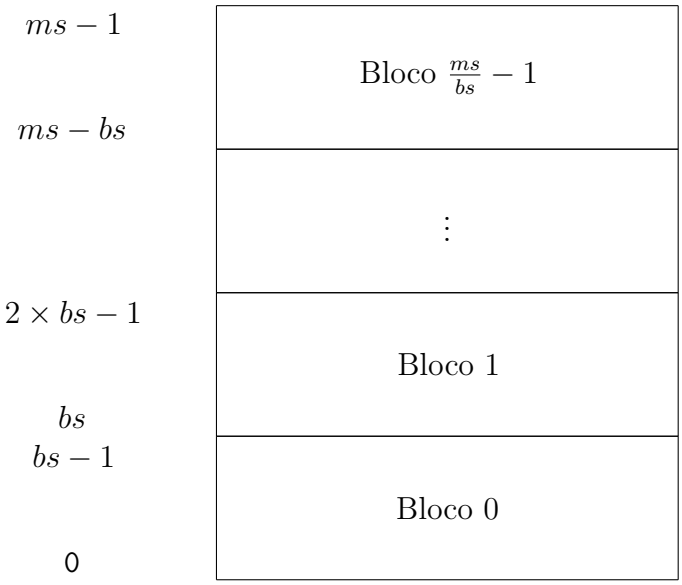

Figura 9 – Divisão da memória de programa em blocos de bs bytes

[1] no caso do presente trabalho, cada palavra é de apenas um byte, e portanto, utiliza-se esses termos indiferentemente

Laço de Cálculo – Caminho pseudoaleatório. Similarmente a outros algoritmos de verificação de integridade como Seshadri et al. (2004), em cada rodada de cálculo i, utiliza-se o PRNG para sortear aleatoriamente um bloco bl da memória de programa (l. 5 do alg. 1). Uma vez selecionado o bloco bl, seus bs bytes de conteúdo são recuperados da memória de programa $\mathscr{M}_{\mathcal{P}}$ em ordem crescente do endereço. Nota-se que os endereços de memória de um bloco arbitrário x variam de $x \times bs$, inclusive, para $(x+1) \times bs$, exclusive. No algoritmo CalcAlg os valores recuperados da memória de programa são temporariamente gravados em um *buffer* v.

Os blocos $bl_1, bl_2, \ldots, bl_N$ selecionados a cada iteração formam um caminho de pseudoaleatório através da memória de programa, o qual é definido pela sequência de números sorteados pelo PRNG e, em última instância, pela semente s, recebida como parte do desafio $\mathcal{C}$ e utilizada para iniciar o estado do PRNG. Consequentemente, o Disv não conhece, previamente ao recebimento do desafio $\mathcal{C}$, em que ordem ocorrerá a visitação dos blocos de memória de programa de forma a atualizar o valor h. Por conseguinte, em caso de o Disv ser malicioso, a cada iteração, o software malicioso $\tilde{\mathbf{S}}$ deve testar se o endereço sob consulta foi modificado e dessa forma adicionar uma instrução de desvio condicional, o que irá impactar no tempo de execução do cálculo de *checksum* (SESHADRI et al., 2004), bem como na contabilização de ciclos de processamento utilizados.

Laço de cálculo – Atualização de h. A atualização do valor de *hash* h é feita como um encadeamento binário de *hash*. A cada rodada de cálculo i, um novo valor de *hash* h_i é calculado, utilizando-se como parâmetros prévio valor de h, h_{i-1}, e o valor do buffer v preenchido com os valores recuperados de memória recuperados a partir do bloco bl_i sorteado (fig. 10 e ll. 6 e 7 do alg. 1). Após um número N suficiente de iterações, o valor calculado h_N leva em consideração o conteúdo de cada um dos blocos de memória, nos quais a memória de programa é dividida.

Laço de cálculo – Critério de parada. O laço para o cálculo do resultado h termina quando um determinado número de iterações predeterminado N é alcançado. Apesar de N não ser diretamente parte do desafio $\mathcal{C}$, é utilizado um valor chamado critério de parada p, este sim parte de $\mathcal{C}$, de forma a testar se o cálculo do valor h está finalizado ou não. p é um prefixo do valor final esperado de h, h_N. Assim sendo, em cada rodada de cálculo i, o algoritmo testa se o cálculo de h foi finalizado testando se p é um prefixo de h_i (l. 4 do alg. 1). A utilização do critério de parada foi pensado de forma que o Disv não saiba de antemão o número de iterações necessárias para o cálculo de h, sem consumir os dados do PRNG e atualizar o valor do *hash*.

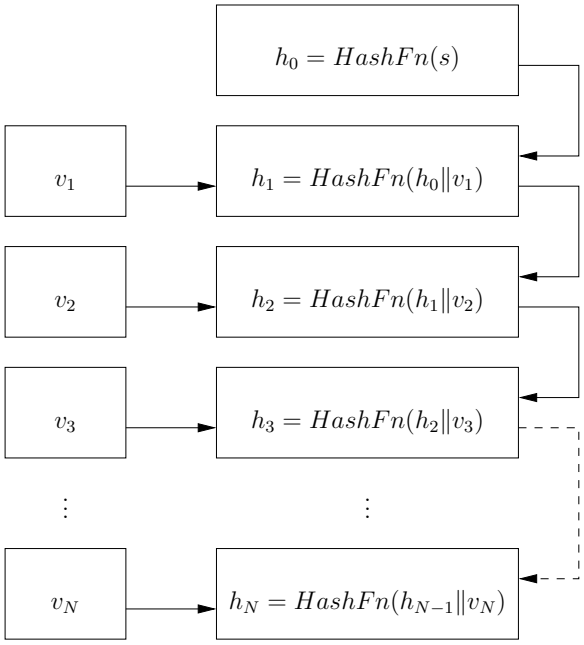

Figura 10 – Cadeia de hash para o cálculo de h. A cada iteração, o valor temporário h_i é calculado a partir de seu valor anterior h_{i-1} e o valor v_i recuperado a partir da memória.

Finalização – Contabilização de ciclos. Adicionalmente, o algoritmo CalcAlg também contabiliza o número de ciclos de *clock* de processamento consumidos no cálculo de h. De forma a contabilizar o número de ciclos de clock, o algoritmo faz uso de uma instrução chamada `CycleCount` fornecida pela plataforma do Disv (ll. 1 e 9 do alg. 1). `CycleCount` fornece uma contabilização da quantidade de ciclos de *clock* desde a inicialização da plataforma. Após o término do cálculo de h, calcula-se o número de ciclos de clock c subtraindo o número de ciclos antes do cálculo, c_0, do número de ciclos após sua execução, c_f (l. 10 do alg. 1). Após definir o número de ciclos, o Disv envia c e h para o verifiifer como resposta do CalcAlg: $\mathcal{R} = (h, c)$.

O algoritmo de cálculo deve ser determinístico, isto é, para mesmos parâmetros de entrada, deve-se retornar os mesmos resultados na saída, o valor de *hash* e o número de ciclos utilizados no cálculo. Isto será necessário, conforme será visto na etapa de Classificação, para comparar o resultado de um Disv potencialmente malicioso, com o de um Disv comprovadamente íntegro. A saída dos dois resultados deve ser similar de forma a atestar que Disv sob análise é íntegro.

4.2.2 Desafio

O Desafio é, cronologicamente, a primeira fase do esquema de verificação de integridade, sendo responsável por gerar os parâmetros de desafio que serão utilizados no algoritmo de cálculo (seção 4.2.1). O desafio é uma ênupla composta por três valores:

$\mathcal{C} = (s, p, bs)$. O critério utilizado para escolha desses três valores é discutido na presente seção e o algoritmo 2 apresenta a sequência de passos a serem discutidos.

Algoritmo 2 Algoritmo da fase DESAFIO (ChallAlg)

1: $s \leftarrow$ `RandomInt`()
2: $bs \leftarrow$ `ChooseBlockSize`($[1, ms]$)
3: `assert`($ms \mod bs = 0$)
4: $N \leftarrow$ `ChooseNumberOfRounds`($ms \div bs$)
5: $p \leftarrow$ `CalculateStopCriteria`(s, n)

Geração da semente. O primeiro passo é a geração de uma semente aleatória. A semente é um inteiro e, como visto na seção 4.2.1, é utilizada na inicialização do Gerador de Números Pseudoaleatórios, e seu tamanho em bytes é definido pelo tamanho do argumento aceito na inicialização do PRNG (l. 3 do alg. 1). É importante que o a semente seja gerada de forma verdadeiramente aleatória, assim um Disv potencialmente malicioso não pode prever seu valor, e, portanto, não pode prever em que ordem se dará o cálculo de *checksum*, o que previne o ataque de pré-computação (seção 2.3).

Tamanho do bloco de memória. O próximo parâmetro a ser definido é o tamanho em palavras do bloco de memória, *bs*. *bs* define quantas posições contíguas de memória serão lidas a cada iteração do algoritmo de cálculo CalcAlg. Os blocos podem ser tão pequenos quanto apenas um byte, caso em que cada posição de memória é considerada um bloco ou, em outro extremo, o bloco pode ser definido tão grande quanto toda a memória de programa. Este último caso é altamente indesejado, uma vez que a imprevisibilidade do cálculo é perdida: em cada iteração do CalcAlg, apenas um bloco (englobando toda a memória) poderia ser sorteado. Por outro lado, a definição de um tamanho pequeno impacta negativamente na quantidade de iterações necessárias do CalcAlg de forma a cobrir toda a memória de programa. Portanto, a definição do tamanho de bloco deve ser feita de forma a ter uma boa variabilidade da execução, bem como ter um tempo de execução do algoritmo razoável.

O tamanho do bloco deve dividir a memória em partes de mesmo tamanho de forma completa e sem interseções. De forma completa quer dizer que qualquer endereço de memória pertence a um bloco. Sem interseções significando que qualquer posição de memória pertence a um único bloco. Para atender a essas restrições, *bs* deve ser um divisor do tamanho da memória, o que está indicado no algoritmo como o teste `assert` (l. 3 do alg. 2). Como geralmente escolhe-se o tamanho de memória como uma potência de 2 há um grande número de opções a ser escolhido como tamanho do bloco.

Número de rodadas. Após a definição do tamanho do bloco, um número suficiente de rodadas de cálculo N do CalcAlg deve ser definido. Conforme definido na

seção 4.2.1, cada rodada de cálculo é composta pelos passos necessários a atualização do valor de hash h. O número de rodadas define, então, quantos blocos de memória serão sorteados pelo CalcAlg e quantas vezes o valor de h será atualizado. Como o Laço de Cálculo em CalcAlg é responsável pelo maior tempo de execução no protocolo, pode-se dizer que N, define, em última instância, o tempo de execução de todo o esquema de verificação de integridade.

Dois objetivos conflitantes da definição de N devem ser levados em consideração: o número de rodadas N deve ser pequeno o suficiente para que o esquema de desafio-resposta execute em um tempo razoável; em contrapartida N deve ser grande o suficiente para que todos os blocos de memória sejam visitados ao menos uma vez. Esta última exigência é importante, pois, ao visitar ao menos uma vez, tem-se a garantia de que uma modificação de valor de qualquer byte da memória de programa irá influenciar o cálculo de h.

Similarmente a outros trabalhos da literatura como Seshadri et al. (2004) e Seshadri, Luk e Shi (2005), utiliza-se dos resultados do problema do coletor de cupons (FERRANTE; SALTALAMACCHIA, 2014) para definir um número mínimo de iterações. O problema do coletor de cupons tem esse nome pela analogia que se faz com a seguinte pergunta: "quantos cupons deve-se obter de forma a completar um álbum?". No caso apresentado no presente trabalho pode-se traduzir o referido problema para: quantos blocos devem ser sorteados aleatoriamente de forma que todos os blocos em que a memória é dividida sejam sorteados ao menos uma vez. Nota-se que um mesmo bloco pode ser sorteado mais de uma vez em diferentes rodadas de cálculo. Como um bloco é sorteado em cada rodada do CalcAlg, os resultados provindos do problema do coletor de cupons permite definir o número de rodadas mínimas requeridas para que cada posição de memória ou bloco seja visitado ao menos uma vez, com alta probabilidade.

Conforme Ferrante e Saltalamacchia (2014), o número esperado de tentativas ou, neste caso em particular, o número de blocos de memória sorteados, $E(T)$, para que todo bloco de memória seja visitado ao menos uma vez é dado pela eq. (4.1). Pela eq. (4.1), também percebe-se que quanto maior o tamanho de bloco escolhido, menos sorteios deve-se fazer de forma a cobrir a memória com alta probabilidade.

$$\begin{aligned} E(T) &= m \sum_{i=1}^{m} \frac{1}{i} \\ &= mH_m \\ &= \frac{ms}{bs} H_{\frac{ms}{bs}} \end{aligned} \tag{4.1}$$

Em que:

a) $m = ms/bs$, ou seja, o número de blocos no qual é dividida a memória;

b) ms = tamanho (em palavras) da memória;

c) bs = tamanho (em palavras) do tamanho do bloco;

d) H_i = o i-ésimo número harmônico.

Critério de parada. Após a definição de s, bs e N, o Verificador calcula o parâmetro de critério de parada p, o qual faz parte do desafio $\mathcal{C}$ que será enviado ao Disv. Conforme explanado na seção 4.2.1, p é um prefixo de ps bytes do valor esperado do hash h_N a ser retornado pelo CalcAlg como resposta ao desafio. Assim sendo, para calcular p, o Verificador necessita calcular h_N. Como o Verificador tem acesso ao conteúdo de memória esperado no Disv o critério de parada pode então ser calculado através do alg. 3.

Algoritmo 3 Cálculo do parâmetro critério de parada p

Entrada(s): 1. A semente, o número de rodadas e o tamanho do bloco: (s, N, bs); 2. O conteúdo esperado da memória de programa de um Disv íntegro $\mathcal{P}^{\checkmark}$: $\mathscr{M}_{\mathcal{P}}$; 3. O tamanho do prefixo p em bytes: ps.

Saída(s): O parâmetro critério de parada p.

1: `Prng.Seed`(s)
2: **para** $i = 1$ até N **faça**
3: $\quad bl \leftarrow$ `Prng.Get`$(\texttt{max} = ms/bs)$
4: $\quad v \leftarrow \mathscr{M}_{\mathcal{P}}[(bl \times bs)...[(bl + 1) \times bs]]$
5: $\quad h \leftarrow$ `HashFn`$(h \| v)$
6: **fim para**
7: $p = h[0..ps]$

O algoritmo para o cálculo do critério de parada p é similar ao CalcAlg pois é necessário calcular o valor hash esperado no cálculo h_N. O laço não condicional **enquanto** é substituído por um laço contável do tipo **para** que varia de de acordo com o número de rodadas: de 1 a N, inclusive (l. 2 do alg. 3). Outra exceção é que, ao invés de consultar sua própria memória, o Verificador dispõe de um cópia do conteúdo de memória esperado para o Disv íntegro $\mathcal{P}^{\checkmark}$. O valor p é, então, selecionado como um prefixo do valor esperado para o hash h_N.

O tamanho do prefixo ps, pode ser selecionado pelo usuário ou técnico que executa a verificação de integridade e deve ter um tamanho ideal: deve ser suficientemente grande de forma que a probabilidade de colisão seja pequena ou negligenciável. A colisão é definida como o caso em que um hash intermediário do cálculo h_i, para qualquer $i \in \{x \in \mathbb{N} | 0 < x < N\}$ tenha o mesmo prefixo do hash resultado final do cálculo h_N, ou

de outra forma: $\{i \in \mathbb{N} | 0 < i < N \wedge h_i[0 \dots ps] = h_N[0 \dots ps]\} \neq \emptyset$. Por outro lado, ps deve ser pequeno o suficiente para que a quantidade mínima de informações do valor esperado de h_N seja enviado para o Disv potencialmente malicioso. Após o cálculo de p, o Verificador pode então enviar o desafio $\mathcal{C} = (s, p, bs)$ ao Disv a ser submetido à verificação de integridade.

4.2.3 Classificação

No estágio de Classificação, o Verificador deve avaliar se Disv analisado é considerado íntegro ou malicioso de acordo com a respostas recebidas. O Verificador analisa ou uma resposta, ou um conjunto de respostas provindas do Disv sob análise e compara os resultados com um conjunto de respostas provindas de um Disv controle ou de referência, o qual é reconhecidamente íntegro. As grandezas disponíveis para a análise são: o *hash* retornado resultado do cálculo (h), o número de ciclos de processamento do algoritmo de cálculo (c) e o tempo de execução do CalcAlg (t). As duas primeiras grandezas são obtidas a partir da execução do algoritmo do resultado do CalcAlg, conforme a seção 4.2.1, enquanto o último valor é obtido no Verificador. Nas próximas subseções, mostra-se como se dá a contabilização de tempo de execução, a captura dos valores de referência ou controle para a análise e, por fim, como se dá a classificação dos resultados e a geração do resultado final do esquema de verificação de integridade.

4.2.3.1 Contabilização do Tempo de execução

O Verificador é responsável por contabilizar o tempo de resposta do Disv a uma requisição de verificação de integridade (alg. 4). Logo após enviar o desafio $\mathcal{C}$, o Verificador inicia um cronômetro de forma a contabilizar o tempo de resposta do Disv (l. 2 do alg. 4). O Verificador, então, espera a execução da etapa de Cálculo pelo Disv e, uma vez que a resposta seja recebida, o cronômetro é finalizado (l. 4) e o tempo de execução (l. 5) é adicionado à resposta $\mathcal{R}$ (l. 6). Dessa forma, adicionalmente aos resultados provindos diretamente do Disv, o tempo de execução também é adicionado a resposta $\mathcal{R}$ e será utilizado para a análise no estágio de Classificação. A esta resposta no qual foi adicionando a contabilização do tempo, chama-se, no presente trabalho, de resposta estendida.

4.2.3.2 Captura das respostas de controle

De forma a capturar o comportamento esperado, o desafio gerado $\mathcal{C}$ é enviado ao Disv sob análise é também enviado para um Disv reconhecidamente íntegro $\mathcal{P}^{\checkmark}$ de forma a capturar a resposta esperada $\mathcal{R}^{\checkmark}$. Observa-se também, que de forma a tornar a classificação mais robusta, várias requisições de verificação de integridade (envio do

Algoritmo 4 Envio de uma requisição de verificação de integridade de um Verificador para o Disv.

Entrada(s): Um desafio $\mathcal{C} = (s, p, bs)$ gerado previamente.
Saída(s): Uma resposta estendida $\mathcal{R} = (h, c, t)$

1: `SendToDisv`($\mathcal{C}$)
2: $t_0 \leftarrow$ `GetTime`() ▷ Inicia o cronômetro
3: $\mathcal{R} \leftarrow$ `WaitForResponse`()
4: $t_f \leftarrow$ `GetTime`() ▷ Finaliza o cronômetro
5: $t \leftarrow t_f - t_0$ ▷ Cálculo do tempo de execução
6: $\mathcal{R} \leftarrow \mathcal{R} \| (t)$ ▷ Formação da resposta estendida

desafio) pode ser emitido ao Disv. Assim, não apenas uma, mas um conjunto de várias respostas pode ser analisado durante a fase de Classificação. Isto é especialmente importante na análise da variável de tempo, já que há uma variação observada no tempo de execução do CalcAlg medido entre diferentes execução. Supondo M requisições são enviadas ao Disv, uma lista de diferentes respostas $[\mathcal{R}_i | 0 \leq i < M]$ são analisadas de forma a decidir se o Disv é íntegro ou malicioso. O número de desafios M é, em cenários reais, limitado apenas pelo tempo disponível para coletar todas as respostas. O Verificador pode então analisar apenas uma resposta, $\mathcal{R} = (h, c, t)$, ou um número arbitrário M de respostas, $\{\mathcal{R}_i = (h_i, c_i, t_i) \mid 0 \leq i < M\}$.

4.2.3.3 Classificação do Disv

A fim de efetuar efetuar sua decisão, o Verificador compara a as respostas provindas do Disv sob análise $\mathcal{P}$, o qual é potencialmente malicioso, e os resultados provindos do Disv confiável $\mathcal{P}^{\checkmark}$, recuperados conforme descrito na seção 4.2.3.2. Após um número suficiente de respostas coletadas de ambos $\mathcal{P}$ e $\mathcal{P}^{\checkmark}$, o algoritmo de classificação, ClassAlg (alg. 5), pode ter início.

Entrada/Saída. O algoritmo de classificação (ClassAlg) recebe duas listas como entrada. A primeira lista A é referente a conjunto de respostas providas do Disv $\mathcal{P}$ sob análise, a qual será testada de forma a definir o resultado do algoritmo de classificação. A segunda lista, B, diz respeito a um conjunto de respostas provindas de um Disv confiável $\mathcal{P}^{\checkmark}$. Ela é utilizada de forma a adaptar o classificador utilizado pelo algoritmo. Opcionalmente, embora não esteja explicitado no alg. 5, um classificador c, previamente adaptado de acordo com a lista B, pode ser recebido como uma das entradas do algoritmo. Ao final da execução do algoritmo, é emitida uma resposta com a afirmação se o Disv sob análise é íntegro ou malicioso.

Teste do valor de hash. Passando à efetiva execução do algoritmo, o algoritmo de classificação, ou ClassAlg, checa se cada resultado de *hash* h_i recuperado de um Disv

Algoritmo 5 Algoritmo do estágio de Classificação (ClassAlg).

Entrada(s): 1. uma lista $A = [\mathcal{R}_i = (h_i, c_i, t_i) | 0 \leq i < M]$ de respostas recebidas de um Disv $\mathcal{P}$ potencialmente malicioso. 2. uma lista $B = \left[\mathcal{R}_i^{\checkmark} = (h_i^{\checkmark}, c_i^{\checkmark}, t_i^{\checkmark}) | 0 \leq i < M'\right]$ de respostas recebidas de um Disv confiável.

Saída(s): Uma resposta afirmando se o $\mathcal{P}$ é íntegro (NEG) ou malicioso (POS).

1: $(h^{\checkmark}, c^{\checkmark}, t^{\checkmark}) \leftarrow$ elemento qualquer de B
2: **para todo** $(h, c, t) \in A$ **faça**
3: **se** $h \neq h^{\checkmark}$ **então retorne** POS
4: **fim se**
5: **se** $c \neq c^{\checkmark}$ **então retorne** POS
6: **fim se**
7: **fim para**
8: $T \leftarrow [t | (h, c, t) \in A]$
9: $T^{\checkmark} \leftarrow [t | (h, c, t) \in B]$
10: $c \leftarrow$ um classificador
11: adaptar c de acordo com os dados de tempo $T^{\checkmark}$
12: result ←resultado da c de para testar os dados T
13: **se** result = "$T \in$ mesma categoria de $T^{\checkmark}$" **então retorne** NEG
14: **senão retorne** POS
15: **fim se**

sob análise é igual ao seu valor esperado (l. 3 do alg. 5). Caso ao menos um valor seja diferente dos valores esperados, e como um valor $h \neq h^{\checkmark}$ é indicador suficiente que o conteúdo de memória foi modificado, então, o Disv sob análise é classificado como malicioso.

Número de ciclos de processamento. Similarmente ao valor de *hash*, o número de ciclos de processamento obtidos do Disv sob análise é comparado ao valor esperado $c^{\checkmark}$ coletado de um Disv confiável $\mathcal{P}^{\checkmark}$. Se algum c obtido não é similar ao valor de controle $c^{\checkmark}$, o algoritmo de cálculo é considerado diferente, o que é um indicativo suficiente de código malicioso, e portanto o Disv sob análise é considerado malicioso (l. 5 do alg. 5). Em casos reais, no entanto, uma pequena variação nos ciclos utilizados é observada mesmo no Disv íntegro $\mathcal{P}^{\checkmark}$, conforme será retratado no capítulo 5. Tais variações devem ser levadas em consideração na análise de ciclos.

Análise do tempo. Se ambos os valores de *hash* e os ciclos de *clock* recuperados de uma consulta de um Disv sob análise estão de acordo com seus valores esperados, uma análise do tempo de execução do algoritmo de cálculo CalcAlg é conduzida (linhas 8 a 15 do alg. 5). Diferentemente das outras grandezas, o tempo de execução não pode ser modificado por um Disv malicioso diretamente, já que o tempo é medido pelo Verificador, que é um dispositivo confiável (seção 2.1).

Um Disv malicioso pode entretanto, propositalmente, atrasar o envio da resposta de seu algoritmo de cálculo malicioso de forma a modificar o seu tempo de execução medido pelo Verificador. Por esta razão, alguns esquemas de verificação de integridade propostos, tais como (SESHADRI et al., 2004; SESHADRI; LUK; SHI, 2005), requerem que o algoritmo de cálculo seja otimizado com relação ao tempo. Caso sejam otimizadas com relação ao tempo, qualquer modificação necessária para efetuar um ataque irá adicionar instruções e, consequentemente, irá impor uma sobrecarga no tempo, que segundo os autores, é detectável a partir da definição de um valor limítrofe aceitável.

Não limita-se a escolha de classificadores a serem utilizados para analisar a distribuição da grandeza de tempo de resposta. Um esquema simples de classificação, que será mostrado no capítulo 5, leva em consideração valores limítrofes simples (t_{baixo}) e (t_{alto}), gerados a partir da distribuição da grandeza tempo retirados das respostas provindas do Disv íntegro ($T^{\checkmark}$). O classificador pode calcular a resposta verificamento se o tempo $t \in T$ coletado a partir do Disv sob análise está contido entre esses dois limites: $t_{\text{baixo}} \leq t \leq t_{\text{alto}}$. Adicionalmente, a distribuição do conjunto de dados tempo recebida a partir o Disv pode ser analisada de uma só vez pelo classificador e, dessa forma, não apenas cada resposta, mas as estatísticas da distribuição, como média e desvio padrão, pode ser utilizada por um classificador. Diversos classificadores também podem ser combinados de forma a calcular a resposta final do esquema de verificação de integridade, de forma a analisar vários aspectos da distribuição de tempo das repostas.

5 IMPLEMENTAÇÃO E VALIDAÇÃO

Nesta seção, é apresentada a configuração de experimentos discutidos para implementar o esquema de verificação de integridade proposto no capítulo 4. Na presente configuração, o Disv foi implementado numa plataforma de prototipagem eletrônica de forma a simular o comportamento de sistemas embarcados. Diferentemente, o Verificador foi implementado utilizando um PC comum, como um *software* implementado em *Python* (Python Software Foundation, 2017). Pretende-se com a finalização do presente trabalho, disponibilizar publicamente as implementações de ambos os sistemas de forma a fomentar a utilização de métodos mais seguros de verificação de integridade de software em casos de mundo real.

5.1 IMPLEMENTAÇÃO

Conforme a seção 2.1, o mecanismo de verificação de integridade tem dois atores: o Verificador e o Disv. Na presente seção é apresentada a implementação dessas duas entidades a realização dos experimentos.

5.1.1 Disv

O Disv foi construído sobre uma plataforma *Arduino Uno* (Arduino, 2017). O Arduino é uma plataforma de prototipagem de software amplamente disponível e também tem uma comunidade de desenvolvedores atuante (EVANS, 2011). A linguagem no cerne do Arduino é a amplamente utilizada linguagem de programação C/C++, contando também com a biblioteca padrão `avrlibc` (EVANS, 2011; RENNA et al., 2013). Por esses motivos, a plataforma foi escolhida para implementação do protótipo, já que sua ampla disponibilidade favorece a reprodutibilidade dos experimentos.

O Arduino também possui as características necessárias para a implementação do algoritmo de verificação de integridade, entre as quais a possibilidade de leitura da memória de programa. A plataforma utiliza o microcontrolador *ATmega328P* (Atmel, 2015), o qual possui arquitetura Harvard. A biblioteca avr-libc (AVR Libc, 2016) utilizada como biblioteca padrão para o referido microcontrolador disponibilizar o módulo *<avr/pgmspace.h>: Program Space Utilities*, o qual permite o acesso de leitura da memória de programa, através de sua instrução `memcpy_P`. Dessa forma, um dos requisitos para a implementação do algoritmo de Cálculo do Disv é atingido.

Outra característica essencial para a implementação do esquema é a possibili-

dade de a plataforma ler os seus próprios ciclos de processamento. O Arduino possui um registrador especial `TCNT1`, o qual contabiliza o número de tiques de *clock* do sistema (AVRbeginners.net, 2017). Tal registrador possui 16 bit o que permite a contabilização de até 65 536 ciclos de *clock*. O processador do Arduino por sua vez, possui uma frequência de 16 MHz, Dessa forma em aproximadamente 4,096 ms ocorre um *overflow* do registrador `TCNT1`, tal evento causa o disparo de uma interrupção. Essas interrupções são capturadas e tratadas por uma rotina que incrementar uma variável `overf_`, previamente inicializada em 0. Assim, através da leitura do registrador `TCNT1` bem como da quantidade acumulada em `overf_`, pode-se contabilizar o número de ciclos de *clock* gastos durante uma consulta de verificação de integridade.

O Disv foi implementando como um simples instrumento de medição, contando com um sensor de luminosidade e um mostrador para exibição do resultado (fig. 11). A programação do Arduino se dá através de duas funções `setup` e `loop`. No `setup` são programadas as operações de inicializações necessárias após a energização ou *reset* do Arduino. Nessa função há as configurações iniciais do instrumento: a configuração da velocidade de comunicação serial, configurações de luz do mostrador, e configurações de *timer*s para o disparo de operações periódicas. Na função `loop`, tem-se as operações periódicas que ocorrem após a execução do `setup`: a leitura periódica do sensor de luminosidade, a indicação de seu valor e a leitura de possíveis dados recebidos através da porta de comunicação serial. O programa implementando ocupa 12 268 bytes da memória de programa, representado 37,4 % do total, e 1338 bytes da memória de dados, representado 67,8 % do total disponível.

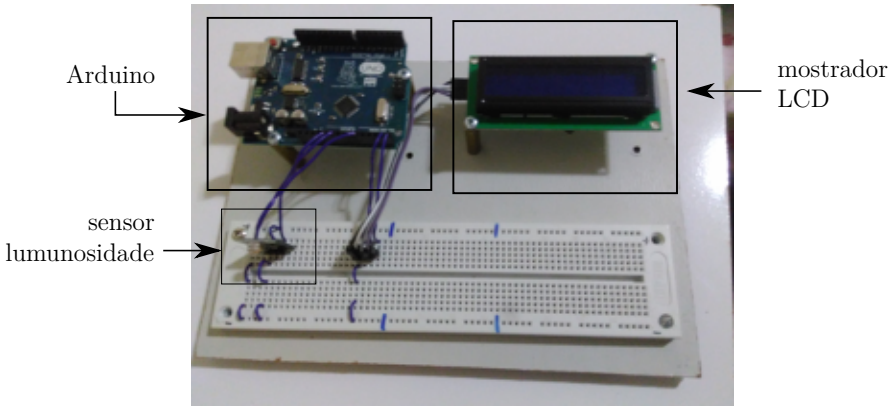

Figura 11 – Implementação do Disv utilizando a plataforma Arduino

5.1.2 Verificador

O Verificador foi implementado como um software escrito em python (Python Software Foundation, 2017) e sua plataforma é um PC comum. O uso de um PC comum favorece a reprodutibilidade dos experimentos e a escolha de Python como linguagem de

programação foi dada porque ela é conveniente em vários aspectos necessários na implementação do verificador, desde bibliotecas disponíveis para a comunicação serial entre o Verificador e o Disv, até bibliotecas para a análise de dados utilizados na classificação dos resultados.

5.1.2.1 Protocolo de comunicação

A comunicação foi implementada de forma síncrona: ao enviar uma mensagem, o Verificador espera até o retorno da mesma para continuar o seu processamento. Conforme mencionando no capítulo 4, as duas mensagens principais são trocadas entre as entidades: a mensagem desafio e a mensagem de resposta. O pacote de transmissão das mensagens possuem os seguintes campos: `STX`, para delimitar o início do pacote transmitido; `CMD`, define qual comando irá trafegar; `FMT`, define qual a representação binária utilizada na área de `payload`; `size`, define o tamanho da área de `payload`; `payload`, utilizado para trafegar a informação desejada; e `CRC`, para a detecção de erros de transmissão (fig. 12). O protocolo trata os inteiros no formato *big-endian*.

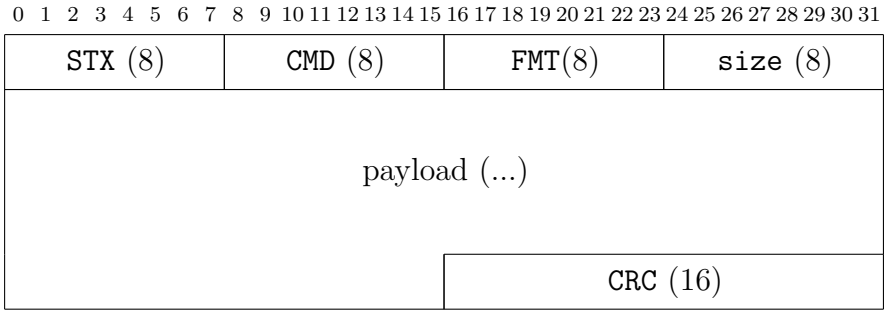

Figura 12 – Estrutura das mensagens entre o Disv e o Verificador. Entre parênteses é descrito o tamanho em bits de cada campo. O campo `payload` tem tamanho variável.

As duas mensagens de interesse e que trafegam pelo canal de comunicação são a mensagem de desafio, `PreffixHashMessagePayload` (fig. 13) e a mensagem de resposta, `TimeAwareHashResponsePayload` (fig. 14). A mensagem `PreffixHashMessagePayload` possui 4 campos: os campo `block_size`, `seed` e `prefix` correspondem aos parâmetros de desafio apresentados no capítulo 4, respectivamente bs, s e p; o campo `memory_size` foi utilizado para testes de implementação e não seria utilizado em uma situação prática. Obseva-se que os campos `seed` e `block_size` tem tamanho fixo de 32 bits, enquanto o tamanho do prefixo é arbitrário e definido ultimamente pelo usuário do Verificador, conforme considerações apresentadas na seção 4.2.2.

O quadro de resposta possui três campos: `hash_bytes`, `diff` e `overflow`. O primeiro campo é utilizado para retornar o resultado final calculado do *checksum*

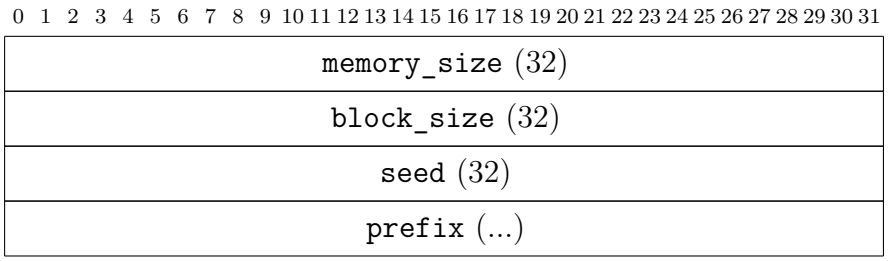

Figura 13 – Mensagem `PreffixHashMessagePayload`, representando um desafio de verificação de integridade $\mathcal{C}$. Entre parênteses está representando o tamanho de cada campo em bits. O campo `prefix` tem tamanho variável.

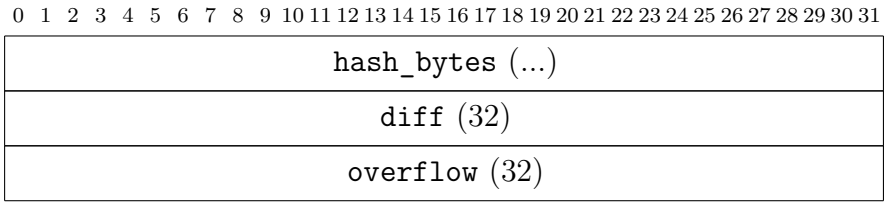

Figura 14 – Mensagem `TimeAwareHashResponsePayload`, representando um resposta $\mathcal{R}$ a um desafio de verificação de integridade. Entre parênteses está representando o tamanho de cada campo em bits. O campo `hash_bytes` tem tamanho dependente do algoritmo *hash* utilizado.

representando o conteúdo de memória, tem o tamanho definido pelo algoritmo de *hash* utilizado. No caso, utilizou-se um algoritmo `Sha1` (NIST, 2012), o qual retorna um valor *hash* de 20 bytes e possui uma implementação disponível para Arduino (Cathedrow, 2010). Os campos `diff` e `overflow` informam os ciclos de *clock*. O `diff` indica a diferença entre o valor inicial, recuperado no início da etapa de Cálculo, e final, recuperado ao final da etapa de Cálculo, do registrador de contabilização de números de *clock*s `TCNT1` de 16 bits, que contabiliza o número de ciclos de máquina do Arduino desde a sua inicialização – de forma a implementar as ll. 1 e 9 do alg. 1 pág. 43. O `overflow`, por sua vez, contabiliza o número de estouros (*overflow*s) da contagem do registrador `TCNT1` ocorridos durante o cálculo do *hash*. A partir desses dois valores, pode-se calcular o número de ciclos consumidos (`clock`s) na etapa de cálculo como na eq. (5.1):

$$\texttt{clock}\text{s} = \texttt{diff} + 2^{16} \times \texttt{overflow}. \tag{5.1}$$

Estabelecida a forma de comunicação entre as duas entidades, pode-se obter pensar nos desafios a serem gerados bem como na organização das respostas obtidas a partir de forma a classificar o dispositivo sob análise.

5.1.3 Definições

Primeiramente é definido o tamanho do prefixo que será utilizado como critério de parada. Foi arbitrado, para os testes realizados, o tamanho de 6 bytes para o prefixo de forma. Dessa forma um potencial atacante somente tem o conhecimento dos 6 primeiros bytes do resultado esperado após o recebimento da consulta de verificação de integridade. Como a função *hash* utilizada nos experimentos produz um valor de 20 bytes como resumo, o atacante não tem conhecimento de 14 bytes da resposta que necessitam ser calculados através do algoritmo de cálculo apresentado na seção 4.2.1. O tamanho do prefixo pode também ser adaptado de acordo de acordo com o desafio escolhido.

Outra preocupação levada em consideração na definição da quantidade de bytes do é a probabilidade de colisão. Por colisão entende-se a situação em que algum valores intermediários do cálculo do valor h possui o mesmo prefixo do valor final esperado, conforme definido na seção 4.2.2. A probabilidade aumenta como o número de rodadas N do algoritmo de cálculo. Considerando a distribuição do valor de *hash* como uniforme e aleatória, pode-se considerar a probabilidade de colisão em uma rodada i como na eq. (5.2):

$$P(A_i) = \left(\frac{1}{2}\right)^{8ps} \tag{5.2}$$

E a probabilidade de colisão em qualquer rodada como na eq. (5.3):

$$\begin{aligned} P(A) &= P(\cup_{i=1}^{N} A_i) \\ P(A) &\leq \sum_{i=1}^{N} P(A_i) = \sum_{i=1}^{N} \left(\frac{1}{2}\right)^{8ps} \\ P(A) &\leq N \left(\frac{1}{2}\right)^{8ps} = \frac{N}{2^{8ps}} \end{aligned} \tag{5.3}$$

Em que:

a) A_i = evento de colisão na rodada i;

b) A = evento de colisão em qualquer rodada;

c) N = número de rodadas;

d) ps = tamanho do prefixo em bytes.

Uma forma de contornar o problema de uma possível colisão é fazer com que o próprio algoritmo que gera o critério de parada (alg. 3, pág. 49) testar se há algum

valor intermediário que gera uma colisão. Nesse caso, o próprio algoritmo pode aumentar deliberadamente o tamanho do prefixo a ser passado como critério de parada.

Outro ponto definido para a implementação foi a escolha da utilização de um gerador de números pseudoaleatório compartilhado entre o Disv e o Verificador. Embora o Arduino possua um gerador de números pseudo aleatórios próprio, não foi encontrada uma boa documentação a respeito de sua implementação. Dessa forma, como o mesmo gerador de números aleatórios deve também ser implementado na outra parte do esquema de verificação: o Verificador. Optou-se por fazer uma implementação de um gerador de números aleatórios do tipo Gerador Congruente Linear (PARK; MILLER, 1988). Utilizou-se também parâmetros já consagrados pela sua utilização no algoritmo de geração de números aleatórios `rand` do biblioteca GLibC (Gnu, 2017). Combinando os mesmos algoritmos e os mesmos parâmetros mesmos algoritmos para geração de números aleatórios foram utilizados em ambos os dispositivos.

5.2 RESULTADOS

Com os dois dispositivos implementados, passou-se à captura de dados de forma a analisar o comportamento das distribuições das variáveis de retorno e de forma a verificar a possibilidade de se ter bons classificadores para delimitar se o dispositivo sob análise é íntegro ou não. As respostas são tratadas e organizadas em uma estrutura de dados tabular para posterior análise, cada resposta vinda do classificador possui um hash, a contabilidade dos ciclos de clock e o tempo de resposta.

Para os testes iniciais limitou-se artificialmente o tamanho da memória de programa para 4096 bytes, o tamanho escolhido limita a verificação de integridade somente a uma primeira área inicial da memória de programa. Por enquanto, os testes foram realizados nessa pequena área por questões meramente práticas, de forma a evitar inconsistências devido lixo de memória deixados por softwares previamente gravados no Arduino, e, dessa forma, garantir que todos os endereços consultados outros programas possivelmente previamente gravados na plataforma ficam dentro do software programado na plataforma. Também evitou-se a área em que fica gravado o *bootloader*, a qual poderia modificar dependendo da plataforma utilizada e conter proteções contra sua leitura. Em casos reais, no entanto, com a área não utilizada preenchida de forma aleatória e sem a figura de um *bootloader* fixo inacessível para leitura, toda a memória de programa do Arduino Uno (de 32 KiB) deve ser utilizada para a verificação de integridade. Dessa forma, apesar dessas limitações artificiais, os testes não perdem a generalidade.

O próximo passo foi a escolha dos parâmetros para a consulta. Para os testes,

foi escolhido uma semente arbitrária qualquer e optou-se pela semente simples, $s = 1$. O tamanho do bloco utilizado foi escolhido como $bs = 32$, dessa forma a memória de programa foi dividida em $m = ms/bs = 128$ blocos. Assim sendo, o número de rodadas da fase de cálculo foi calculado de acordo com o problema do Coletor de Cupons (seção 4.2.2) e definido como $N = 696$, conforme eq. (5.4):

a partir da eq. (4.1) (pág. 48):

$$E(T) = mH_m$$

substituindo H_m pela aproximação definida em Sondow e Weisstein (2017):

$$\begin{aligned} E(T) &\approx m(\ln m + \gamma + \frac{1}{2m}) \\ &\approx m \ln m + m\gamma + \frac{1}{2} \end{aligned}$$

como $m = 128$ e $\gamma \approx 0{,}578$, em que γ é a Constante Euler-Mascheroni (WEISSTEIN, 2017)

$$\begin{aligned} E(T) &\approx 128 \ln 128 + 128 \times 0{,}578 + \frac{1}{2} \\ &\approx 695.443 \end{aligned} \tag{5.4}$$

Na tab. 5 são mostradas algumas repostas retornadas por um Disv íntegro. Pode-se ver que para os resultados de tempo de resposta para cada consulta para $N = 696$ é aproximadamente 3,2 s. A partir da pequena amostra de resultados mostrada na tab. 5, pode-se discutir alguns comportamentos da consulta de verificação de integridade. Como era de se esperar os resultados de *hash* retornados pelo Disv são exatamente os mesmos refletindo o mesmo conteúdo não-alterado da memória entre duas execuções do CalcAlg (alg. 1, pág. 43). O comportamento dos valores de tempo de execução também exibe uma variação que deve ser levada em consideração na definição dos classificadores, pois está sujeita a imprecisão da contabilização do tempo no dispositivo Verificador. Outro fenômeno é a variabilidade dos ciclos de *clock* dentre as execuções. Os *clock*s variam de uma execução a outra da verificação, o que pode ser causado por uma imprevisibilidade de interrupções disparadas durante a execução do algoritmo de cálculo.

Para mesmos parâmetros de desafio repetiu-se a consulta de verificação 1024 vezes e observou-se a distribuição dos resultados de ciclo e tempo. Na fig. 15 vê-se a frequência dos **ciclo**s recebidos a partir de várias consultas de verificação de integridade, enquanto que na fig. 16 têm-se a distribuição da variável tempo. Na fig. 17 visualiza-se um diagrama de dispersão dos resultados recebidos do Disv íntegro. Na tab. 6 são mostradas algumas estatísticas referentes à distribuição dos **ciclo**s e do **tempo**.

Tabela 5 – Resultado de uma amostra de 5 consultas de verificação de integridade retornadas de um Disv íntegro. Com os seguintes parâmetros: $bs = 32, s = 1, N = 696$.

	hash	**ciclo**	**tempo**
1	0ad7558c89b8bb040d162ff07708164b2580221d	43757500	3.220
2	0ad7558c89b8bb040d162ff07708164b2580221d	43757559	3.214
3	0ad7558c89b8bb040d162ff07708164b2580221d	43757559	3.214
4	0ad7558c89b8bb040d162ff07708164b2580221d	43757461	3.214
5	0ad7558c89b8bb040d162ff07708164b2580221d	43757559	3.248

Os coeficientes de variação para a distribuição dos valores retornadas de número de ciclos e tempo de execução são mostrados na tab. 6. O coeficiente de variação: definida como a razão entre o desvio padrão e a média e é utilizada para comparar a variabilidade entre duas distribuições distintas (EVERITT; SKRONDAL, 2002). Vê-se que a distribuição dos **ciclo**s, com coeficiente de variação de $\approx 1{,}1 \times 10^{-6}$, é mais homogênea do que a distribuição do **tempo**, com coeficiente de variação 3 ordens de magnitude maior ($\approx 2{,}8 \times 10^{-3}$). O fato de a contabilização dos **ciclo**s não estar sujeita a latência da comunicação ou precisão de relógio utilizada, ao contrário da distribuição da característica **tempo**, favorece esse resultado.

Analisando primeiramente a distribuição da coluna **ciclo** (fig. 15) pode-se constatar que os resultados se agrupam em torno de quatro valores. Apesar de possuir uma variação, no entanto, ela foi considerada desprezível com relação à magnitude da sua média. A distribuição do **tempo** (fig. 16), por sua vez, possui uma variação não desprezível, a qual deve ser levada em consideração nos testes para classificação. Os valores de **tempo** também possuem uma distribuição assimétrica, com, com maior frequência de valores na cauda direita da distribuição.

Tabela 6 – Algumas estatísticas das distribuições de ciclos e tempo

	desvio padrão	média	coeficiente de variação
ciclo	48,955237	4,375752e+07	0,000001
tempo (s)	0,008961	3,217036e+00	0,002786

5.2.1 Ataques implementados

De forma a testar algoritmos de classificação para o presente esquema, foram implementados 2 tipos de ataques baseados no ataque de cópia de memória (seção 2.3): **ataque I** e **ataque II**. O esquema que utilizamos no presente trabalho utiliza um parâmetro aleatório, portanto uma ataque de pré-computação não seria efetivo, já que retornaria o valor de *hash* esperado a cada consulta. O ataque de cópia de memória, por

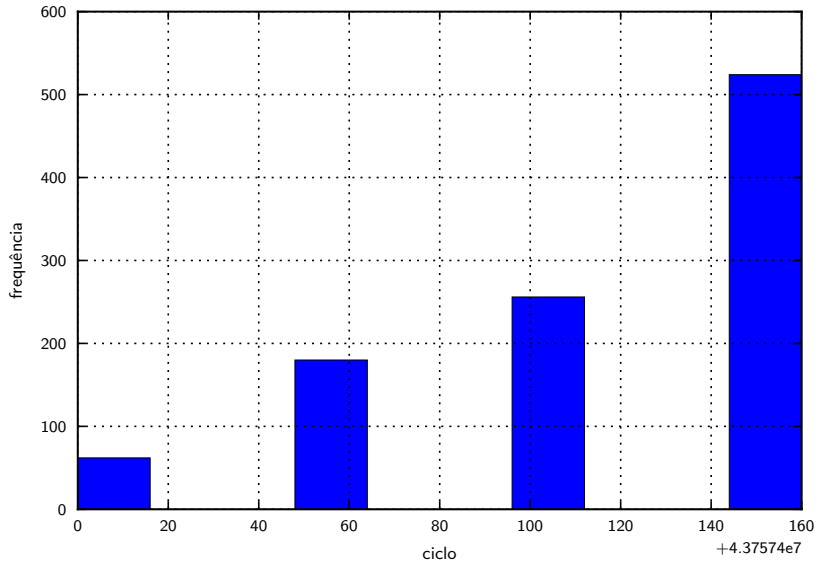

Figura 15 – Histograma representado a distribuição da contabilização do ciclos para 1024 consultas.

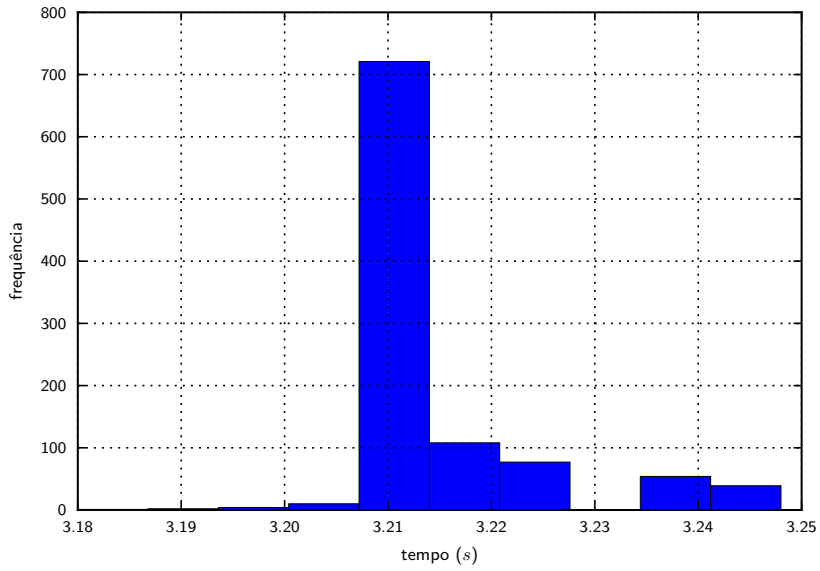

Figura 16 – Histograma representado a distribuição da contabilização do tempo para 1024 consultas.

sua vez, além de retornar o valor de *hash* esperado, é o mais vantajoso, isto é, incorre em menos de tempo de execução adicional, para o atacante: supõe-se que o software pode ser armazenado completamente em uma área não utilizada da memória e, portanto, a operação de redirecionamento das consultas de verificação de integridade para a região onde está armazenado o software íntegro **S** podem ser feitas conforma mostrado na

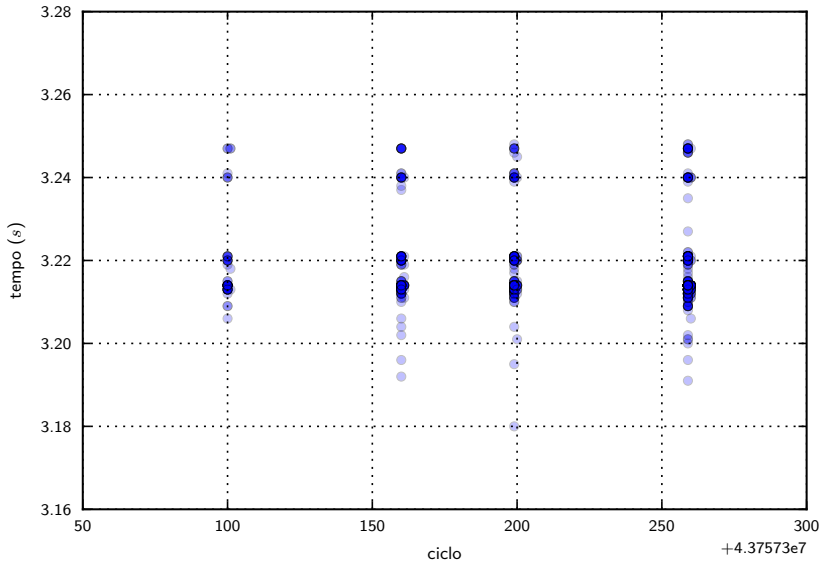

Figura 17 – Diagrama de dispersão para 1024 consultas.

tab. 1 (pág. 28). Dessa forma o impacto no tempo de execução para um ataque será bem menor do que no caso de um ataque mais complexo como o de compressão (seção 2.3) por exemplo.

Primeiramente discute-se o **ataque I**, um ataque de cópia de memória ingênuo já que não tenta reproduzir a quantidade de ciclos contabilizada na implementação do CalcAlg íntegro. Como são adicionadas instruções para modificar o endereço consultado de forma a direcionar a consulta para a área de memória, haverá uma modificação no número de ciclos bem como no tempo da consulta. Observa-se também que esse tipo de ataque pode ser utilizado, com sucesso, para outros mecanismos de verificação de integridade que não analisam o número de ciclos de *clock* do algoritmo de cálculo.

Também foi implementado um **ataque II**, o qual objetiva fraudar o número de ciclos de *clock*. Para a implementação do ataque, o adversário deve contabilizar o número de rodadas, isto é, iterações do Laço de Cálculo (seção 4.2.1), que são utilizados no algoritmo de cálculo. Posteriormente, esse o número de rodadas contabilizado é aplicado à uma equação matemática para a contabilização do número de ciclos utilizados pelo dispositivo de forma a retornar o valor ao final da consulta de verificação de integridade. É preciso levar em consideração que a contabilização dos ciclos não é trivial, visto que não somente o número de rodadas como o tamanho de blocos deve ser levado em consideração. No entanto, considera-se que um atacante habilidoso o suficiente consegue implementar o ataque com sucesso.

5.2.2 Classificação

Para visualizar os resultados dos ataques primeiramente compara-se visualmente os resultados para os ataques propostos. Na tab. 7 são mostrados os primeiros resultados de consulta de verificação de integridade para um dispositivo malicioso no qual foi implementado com sucesso o **ataque I**.

Tabela 7 – Amostra de consultas de verificação de integridade para um dispositivo malicioso que implementa o **ataque I**

	hash	**ciclos**	**tempo** (s)
1	0ad7558c89b8bb040d162ff07708164b2580221d	43763795	3.214
2	0ad7558c89b8bb040d162ff07708164b2580221d	43763795	3.214
3	0ad7558c89b8bb040d162ff07708164b2580221d	43763734	3.213
4	0ad7558c89b8bb040d162ff07708164b2580221d	43763734	3.248
5	0ad7558c89b8bb040d162ff07708164b2580221d	43763735	3.214

Conforme esperado, os valores de *hash* retornados pelo Disv malicioso, com o **ataque I** são os mesmos esperados em uma consulta de verificação realizada considerando um Disv íntegro. Por conseguinte, somente a análise do valor *hash* não é suficiente para detectar um dispositivo malicioso que implementa esse tipo de ataque. No entanto, a dificuldade para o atacante se manifesta ao requerer também que o atacante retorne o número de ciclos de forma a ser analisada pelo esquema. Como o **ataque I** não prevê esse tipo de desafio, o número de ciclos retornado pelo dispositivo atacado é bem diferente do número retornado por uma consulta a partir de um Disv íntegro, conforme pode ser visto no histograma apresentado na fig. 18. Devido as instruções adicionais necessárias à implementação do **ataque I**, pode-se perceber que há uma clara separação entre as distribuição dos ciclos retornados em cada um dos casos.

A mesma diferenciação, no entanto, não se reflete na distribuição do tempo, conforme pode ser visto na fig. 19. A distribuição do **tempo** é similar entre os dois casos, e, visualmente não há diferenciação entre as duas distribuições. E isso reflete em uma dificuldade de classificação entre as duas distribuições baseando-se somente no **tempo**.

De forma a testar a diferenciação entre as distribuições consideramos um classificador que compara as três grandezas, ou características[1], retornadas de uma consulta de verificação de integridade. Conforme seção 4.2.3, a comparação do *hash* é realizada diretamente através de igualdade: se o *hash* for igual é válido, se for diferente o software presente no Disv é fraudulento. No caso dos **ciclo**s a comparação também é similar, embora deve ser levado em consideração a pequena variabilidade. Dessa forma,

[1] *features*

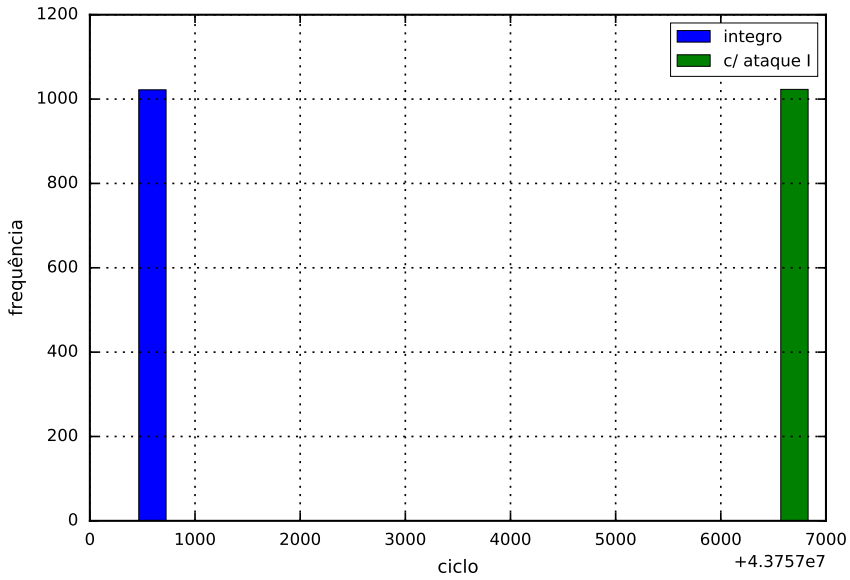

Figura 18 – Histogramas que mostram a distribuição dos ciclos retornadas nos casos de Disv íntegro e sob o **ataque I**.

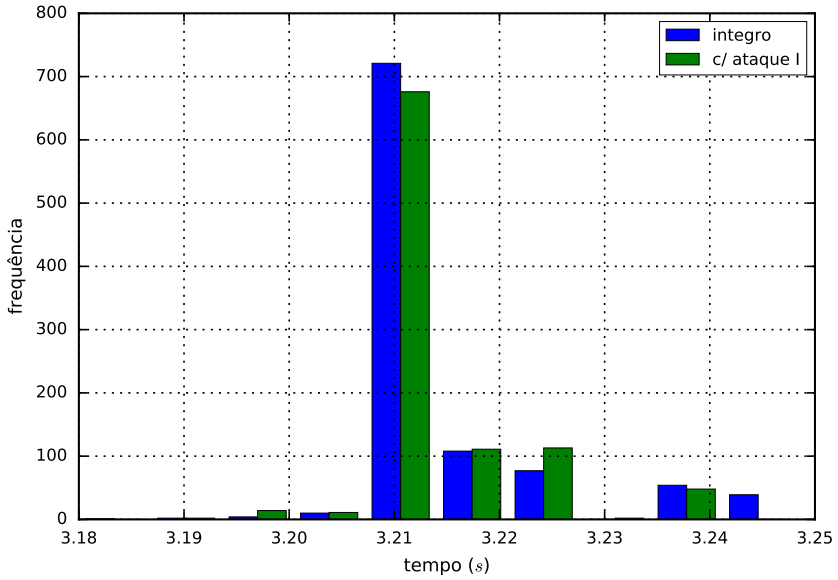

Figura 19 – Histogramas que mostram as distribuições dos tempos retornadas nos casos de Disv íntegro e sob o **ataque I**.

considera-se os **ciclo**s como válidos se eles forem contabilizados dentro do intervalo mínimo e máximo dos **ciclo**s recuperados de várias consultas ao software íntegro.

A classificação do **tempo** é um pouco diferente, já que a sua variabilidade não pode ser desprezada. Primeiramente, os dados foram normalizados segundo a mediana,

como centro da normalização, e o intervalo inter quantil, ou IQR na sua sigla em inglês, como medida de distância. Para tal utilizamos `RobustScaler` (Scikit-learn Developers, 2017) do módulo (PEDREGOSA et al., 2011). Segundo Scikit-learn Developers (2017), esse tipo de normalização é útil na presença de *outliers*, o que é o caso para os dados utilizados no experimento. Na fig. 20 mostra-se a distribuição do tempo normalizada juntamente com a distribuição provinda do ataque.

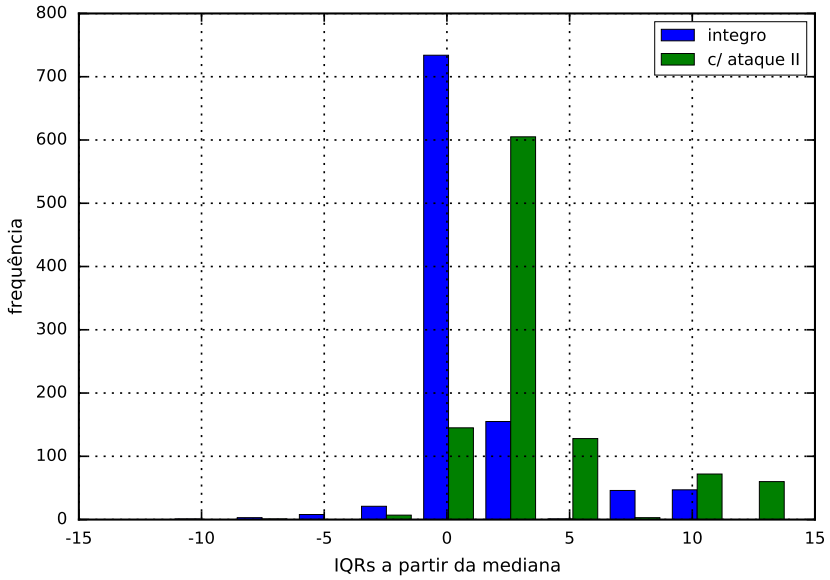

Figura 20 – Histograma do tempo normalizado.

De forma a poder-se classificar os resultados passa-se então a definição do ponto de corte para o qual serão aceitos os tempos. Para a definição do ponto de corte, foram verificadas várias possibilidades. Para tal utilizou-se de uma métrica chamada "Escore F_1", o qual é definida como uma média harmônica entre a sensibilidade e a precisão. A estatística Escore F_1 varia de 0, sendo este o pior valor, a 1, sendo esse o melhor valor. Na fig. 21 tem-se o comportamento da referida métrica para vários pontos de corte testados, especificamente de 0.1 a 10.

Observa-se que o valor do Escore F_1 atinge seu pico por volta por volta do ponto de corte 1. De fato, seu valor nesse ponto atinge o valor máximo, de $F1 \approx 0{,}858$. Para a classificação, utilizou-se o ponto de corte em 1. Assim sendo, utilizando-se de um classificador que verifica o **hash**, os **ciclo**s e o **tempo**, este último dentro de um ponto de corte de 1 IQR para mais ou para menos, efetua-se a tarefa de classificação.

Primeiramente analisa-se o desempenho do classificador na presença de um dispositivo Disv íntegro e um Disv malicioso, o qual implementa o **ataque I**, através da matriz de confusão mostrada na fig. 22. O classificador binário utilizado categoriza

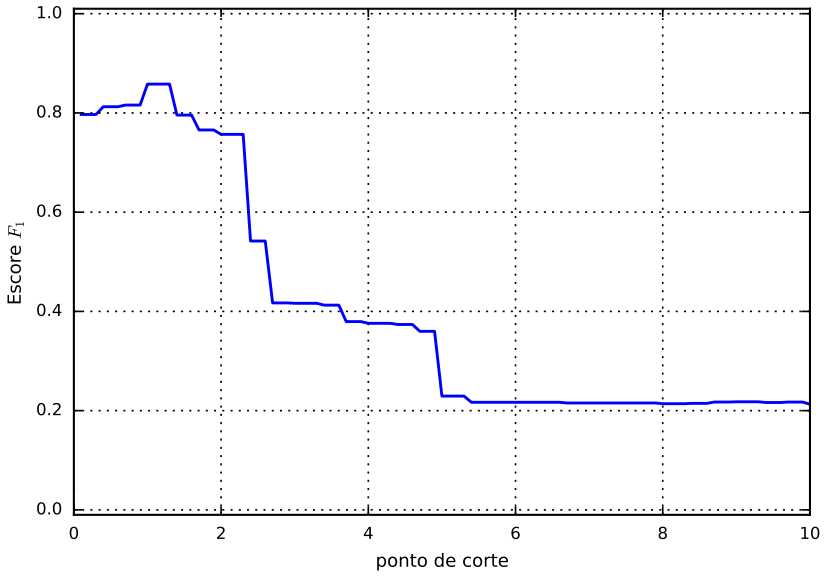

Figura 21 – Escore F_1 para vários pontos de corte do classificador.

em dois estados possíveis: POS, determinando um dispositivo fraudulaneto, ou NEG, determinando um dispositivo íntegro. Devido à análise do **ciclo**, e como o ataque não calcula o número de ciclos esperados, o classificador apresenta uma Sensibilidade (EVERITT; SKRONDAL, 2002) de 100 %, já que classifica todos os resultados provindos do dispositivo fraudulento como tal.

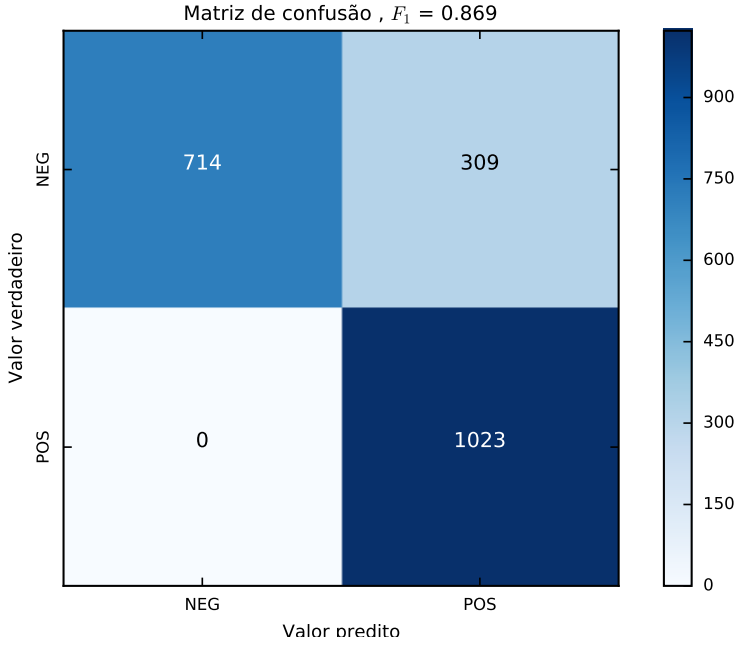

Figura 22 – Matriz de confusão para o caso do **ataque I**.

Para fins comparação, na fig. 23 é apresentada a matriz de confusão para

o mesmo classificador apresentado anteriormente, exceto, no entanto, que não há a classificação por número de número de **ciclo**s. O classificador somente compara os valores das propriedades **hash** e **tempo**. Observa-se que devido aos tempos de execução da implementação íntegra e da implementação com o **ataque I** serem similares, a maioria dos resultados provindos de um Disv fraudulento (valor verdadeiro POS) são classificados como íntegros ou NEG. Nesse caso, obviamente não há uma classificação útil já que há um grande números de falsos negativos, isto é, de dispositivos maliciosos classificados como íntegros.

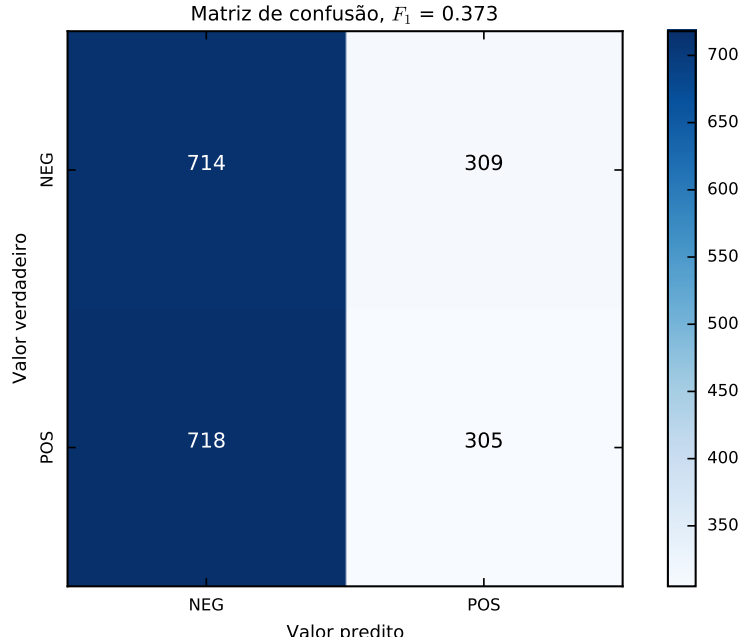

Figura 23 – Matriz de confusão para o caso do **ataque I**, sem classificação por ciclos de clock.

Para o caso do **ataque II**, a necessidade de contabilização do número de ciclos causa um deslocamento da distribuição do tempo de execução para a direita, referente à necessidade de processamento adicional, conforme pode ser visto na fig. 24. Esse deslocamento é suficiente para permitir uma diferenciação mais adequada pelo classificador, conforme a matriz de confusão mostrada na fig. 25. Portanto, apesar de o ataque em questão calcular corretamente o número de ciclos, tem-se que, devido ao custo adicional de tempo envolvido nesse cálculo, a diferença de **tempo** entre o Disv íntegro e malicioso é suficiente para que a Sensibilidade do classificador ainda continue com um valor próximo a 100 % ($= \frac{1001}{22+1001} \approx 0.978$).

5.3 DISCUSSÕES

Através da análise do diagrama de dispersão da fig. 17 (pág. 63), observa-se a dispersão da distribuição do tempo de execução e do número de ciclos retornados

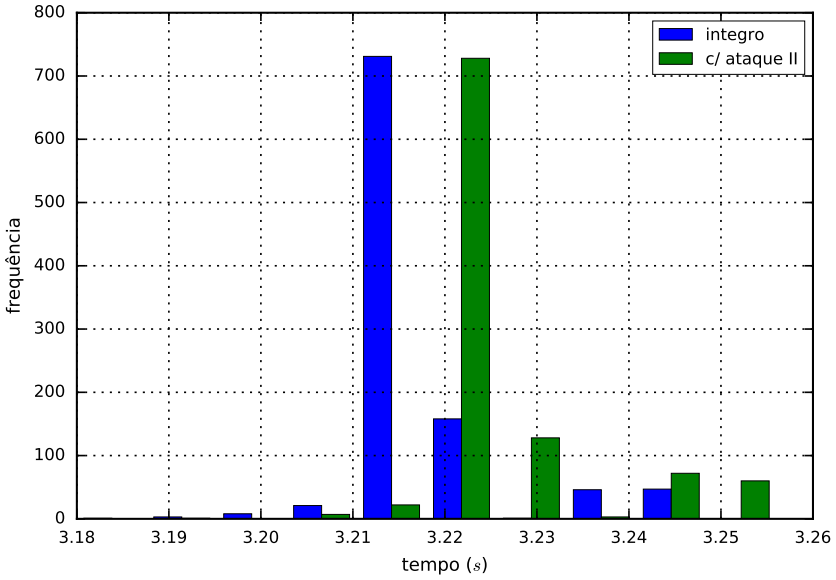

Figura 24 – Comparação dos histogramas dos tempo de execução.

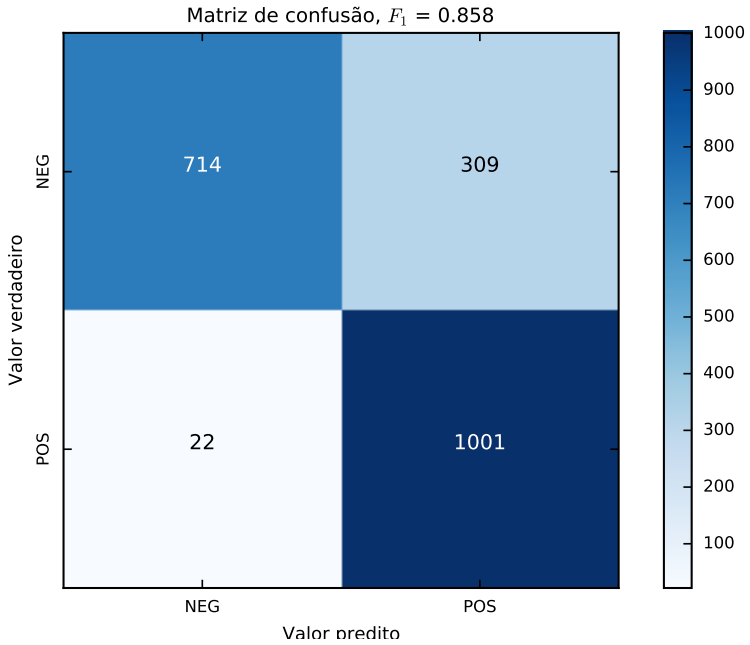

Figura 25 – Matriz de confusão para o classificador na presença do **ataque II**.

por um Disv íntegro. A dispersão do tempo de execução pode ser devida a latência de comunicação entre os equipamentos, principalmente devido à utilização de um PC padrão, já que diversos processos concorrentes podem afetar a contabilização do tempo. O mesmo argumento não pode ser utilizado na variação do número de ciclos retornado à cada consulta enviada ao Disv. Entretanto, aponta-se que as variações podem estar relacionadas à interrupções espúrias durante a execução da rotina de verificação de integridade pelo Arduino. Devido à necessidade de lidar com interrupções para contabilizar a quantidade

de *overflow*s do registrador contador dos ciclos de *clock* (seção 5.1.2.1), não foi possível desabilitar completamente as interrupções da plataforma. A rotina pode ser implementada em outros equipamentos ou plataformas de forma a verificar se o fenômeno de variação do número de ciclos ainda ocorre.

Embora seja fato que um Disv malicioso possa falsear o número de ciclos de *clock* a ser retornado, há um processamento adicional necessário ao Disv malicioso para que possa (i) contabilizar o número de rodadas, (ii) levar em consideração do tamanho do bloco de memória e, por final, (iii) utilizar uma fórmula matemática para verificar o número de ciclos esperado em uma consulta íntegra. Tal processamento adicional necessário é responsável por deslocar a distribuição do tempo de resposta para a direita, conforme pode ser visto na fig. 24. Com isso, um classificador adequado pode detectar essa diferença em tempo de execução, e classificar tais dados como provindos de um Disv fraudulento.

Outro ponto que endossa o deslocamento do tempo de execução no caso de ataque é que o cenário apresentado no presente trabalho é otimista ao atacante, visto que em exemplos do mundo real, o atacante não deve ter espaço suficiente para armazenar uma cópia completa do software íntegro na memória de programa. Isto pode ser forçado pelo preenchimento de espaços não utilizados na memória com valores aleatórios (SPINELLIS, 2000; SESHADRI et al., 2004), os quais também devem ser consultados pela verificação de integridade. Com o preenchimento completo da memória, não seria possível implementar o ataque de cópia de memória simples apresentado na seção 2.3. Ainda assim, poderia ser possível armazenar somente informações diferenciais entre o software íntegro e o software confiável de forma a poder reproduzir o resultado correto do hash esperado em uma consulta de verificação. No entanto, a cada consulta de memória essa estrutura de informações de diferenciação deveriam ser consultadas, o que acarretaria em um consulta mais complexa e mais onerosa (com relação à tempo) do que a apresentada na tab. 1 (pág. 28).

Em trabalhos propostas anteriormente, uma das suas premissas é que a implementação da rotina de cálculo de *checksum* seja feito de maneira otimizada (SESHADRI et al., 2004; SESHADRI; LUK; SHI, 2005; LI; MCCUNE; PERRIG, 2011). No método proposto no presente trabalho, utilizou-se, na implementação do algoritmo de cálculo do valor do resultado h (alg. 1): uma função para recuperação do próximo valor do PRNG, uma função para recuperar valores da memória de programa e uma função para o cálculo do *hash* criptográfico `Sha1`. Sobre a rotina de cálculo de hash `Sha1`, ela é possui definições em termos de operações simples (tais como adições, rotações, deslocamentos, operações lógicas) (EASTLAKE 3RD; JONES, 2001), e sua implementação de maneira

otimizada é um hipótese plausível (CARMO; MACHADO, 2009). Raciocínio similar pode ser utilizada para a operações de geração do próximo número aleatório pelo PRNG, a qual envolve apenas as operações de multiplicação, adição e módulo, bem como na recuperação de valores da memória, a qual é provida pela biblioteca padrão AVR Libc (2016). Dessa forma, parece não ser crível que um atacante possa otimizar as funções de forma a conseguir tempo adicional para calcular os resultados esperados de ciclos de *clock* c bem do *hash* h.

Certamente, não pode-se definir quais tipos de ataque serão lançados contra o esquema de verificação e que, supostamente, um ataque elaborado o suficiente pode efetuar retornar um tempo de execução similar ao tempo de execução do algoritmo de cálculo para um Disv íntegro. No entanto, também não há limitação nos tipos de classificadores que podem ser utilizados, e classificadores elaborados que levam em consideração não somente uma resposta, mas várias respostas retornadas em diversas consultas podem ser utilizados. Tais classificadores poderiam verificar, por exemplo, além de medidas de tendência central, também a dispersão típica de tais valores de forma a montar o resultado da classificação.

6 CONCLUSÃO E TRABALHOS FUTUROS

No presente trabalho, discutiu-se a respeito do problema de verificação de integridade em sistemas embarcados, o qual é um mecanismo no qual um dispositivo confiável Verificador deve inquirir um dispositivo potencialmente malicioso Disv e através de suas respostas definir se o Disv é ou não confiável. Portanto, a verificação de integridade realiza uma auditoria do dispositivo embarcado no Disv, verificando se o mesmo sofreu alterações indevidas ou não. A verificação de integridade tem várias aplicação em cenários reais, em que tais dispositivos críticos operam em ambientes hostis e podem ser alvo de modificação por um adversário ou atacante. Esses sistemas embarcados geralmente possuem restrições de recursos computacionais e custos. Entre esses tipo é o cenário da metrologia legal em que dispositivos de medição podem a vir a ser alvos de fraudadores, o que representa uma ameaça à justa troca comercial.

No trabalho foi apresentado um novo esquema de verificação de integridade baseado em software para sistemas embarcados. Esse tipo de verificação, por ser baseada em software, apresenta uma vantagem por não necessitar de hardware adicional presente na plataforma, o que é vantajoso para cenários de baixo custo. O esquema proposto apresentado é composto por 3 estágios: Desafio, no qual o ocorre a geração do desafio $\mathcal{C}$ de verificação de integridade, Cálculo, no qual ocorre o cálculo de um resultado $\mathcal{R}$ de acordo com o desafio $\mathcal{C}$ e Classificação, no qual ocorre a avaliação do resultado $\mathcal{R}$ e a definição se o dispositivo é ou não íntegro. Para cada um desses estágios, é proposto um algoritmo para implementá-lo. A proposta apresenta uma melhoria aos esquemas propostos na literatura, já que que adiciona uma dificuldade à implementação de ataques: a quantidade de ciclos de instrução (*clocks*) utilizados para efetuar o cálculo da resposta da requisição de verificação deve ser informado, juntamente ao hash esperado h. O resultado da verificação de integridade também é integrado por uma característica da plataforma (ciclos de *clock*). Essas dificuldades adicionais a um adversário tornam difícil a reprodução do resultado da verificação de integridade por um software malicioso em um tempo de execução aceitável.

Apresentou-se um protótipo de Disv utilizando a plataforma de prototipagem eletrônica Arduino (Arduino, 2017) e também um protótipo do Verificador em PC comum, como um software implentado em Python (Python Software Foundation, 2017). Como o Arduino é uma plataforma amplamente disponível e com a disponibilização de tais implementações, pretende-se contribuir para o desenvolvimento da pesquisa na área de verificação de integridade baseado em software.

Com os protótipos implementados, foram conduzidas várias requisições de verificações de integridade e armazenado os resultados das características **hash**, **ciclo**s e **tempo**. Foram analisados os comportamentos dessas características para o caso de um Disv íntegro e um Disv malicioso, com dois tipos de ataques (**ataque I** e **ataque II**) baseados em um ataque de cópia de memória em um cenário otimista ao atacante, em que há espaço suficiente para armazenar uma cópia do software íntegro **S** e, assim, há menos operações adicionais a serem realizadas para a implementação dos ataques. Através da análise do ciclo de *clock*, foi possível detectar a ocorrências do **ataque I**, e através da análise de tempo de execução, foi possível detectar a ocorrência do **ataque II**. Neste último caso, a necessidade de contabilização dos ciclos de execução para a implementação correta do ataque impactou negativamente no tempo de execução, permitindo uma correta diferenciação entre o tempo gasto para executar a verificação de integridade para um Disv íntegro e um Disv malicioso. Com esses resultados, acredita-se que o método proposto pode ser utilizado com sucesso em cenários do mundo real, que possuem características mais restritivas ao atacante.

Como resultado do presente trabalho, o artigo Castro et al. (2017) foi submetido e aprovado para apresentação no congresso PICOM2017 (PICom-2017 Web Team, 2017).

Como trabalhos futuros, pretende-se analisar o comportamento do tempo de execução para rotinas de ataques com tempos similares a rotina íntegra (**S**) proposta. Verificando outros classificadores mais avançados que analisam não somente um resultado, mas um comportamento de vários resultados da verificação de integridade, como a distribuição da característica **tempo**. Para isso classificadores inteligentes baseados, por exemplo, em lógica difusa[1] (KLIR; YUAN, 1995) ou SVMs (HEARST et al., 1998) podem ser utilizados.

Também pretende-se implementar o Verificador de forma embarcada, utilizando também uma plataforma de prototipagem. De forma que a contabilização do tempo de execução não sofra influência de um Sistema Operacional, além de poder utilizar relógios de maior precisão.

Um ponto a notar é a dificuldade em desenvolver bons esquemas de verificação de integridade baseados em software, com alguns trabalhos da literatura como (CASTELLUCCIA et al., 2009), questionando inclusive a possibilidade de desenvolvê-los corretamente. A verificação de integridade em software tem as vantagens de baixo custo e possibilidade de utilização em sistemas legados. No entanto, em cenários novos, ou

[1] *fuzzy*

onde custo é um fator menos importantes, soluções baseadas em hardware podem ser utilizadas, como por exemplo, SGX (COSTAN; DEVADAS, 2016).

REFERÊNCIAS

ACM, I. **ACM Digital Library**. 2017. Disponível em: <http://dl.acm.org/dl.cfm>. Citado na página 32.

AMBROSIN, M. et al. Sana: Secure and scalable aggregate network attestation. In: **Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security**. New York, NY, USA: ACM, 2016. (CCS '16), p. 731–742. ISBN 978-1-4503-4139-4. Disponível em: <http://doi.acm.org/10.1145/2976749.2978335>. Citado 2 vezes nas páginas 36 e 38.

ARBAUGH, W. A.; FARBER, D. J.; SMITH, J. M. A secure and reliable bootstrap architecture. In: **Proceedings. 1997 IEEE Symposium on Security and Privacy**. Oakland, CA, USA, USA: IEEE, 1997. p. 65–71. ISSN 1081-6011. Citado na página 21.

Arduino. **Arduino Uno & Genuino Uno**. 2017. <https://www.arduino.cc/en/Main/ArduinoBoardUno>. Accessed: 2017-03-14. Disponível em: <https://www.arduino.cc/en/Main/ArduinoBoardUno>. Citado 2 vezes nas páginas 54 e 72.

Arm Limited. **Security on ARM TrustZone**. 2017. Disponível em: <https://www.arm.com/products/security-on-arm/trustzone>. Citado na página 35.

ARMKNECHT, F. et al. A security framework for the analysis and design of software attestation. In: **Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security**. New York, NY, USA: ACM, 2013. (CCS '13), p. 1–12. ISBN 978-1-4503-2477-9. Disponível em: <http://doi.acm.org/10.1145/2508859.2516650>. Citado 7 vezes nas páginas 21, 25, 26, 34, 37, 38 e 40.

ASOKAN, N. et al. Seda: Scalable embedded device attestation. In: **Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security**. New York, NY, USA: ACM, 2015. (CCS '15), p. 964–975. ISBN 978-1-4503-3832-5. Disponível em: <http://doi.acm.org/10.1145/2810103.2813670>. Citado 2 vezes nas páginas 36 e 38.

Atmel. **AVR32 – Architecture Document**. 2011. Disponível em: <http://www.atmel.com/images/doc32000.pdf>. Citado na página 41.

Atmel. **ATMEL 8-BIT MICROCONTROLLER WITH 4/8/16/32KBYTES IN-SYSTEM PROGRAMMABLE FLASH**. 2015. Disponível em: <http://www.atmel.com/images/Atmel-8271-8-bit-AVR-Microcontroller-ATmega48A-48PA-88A-88PA-168A-168PA-328-328P_datasheet_Complete.pdf>. Citado na página 54.

AVR Libc. **AVR Libc Home Page**. 2016. Disponível em: <http://www.nongnu.org/avr-libc/>. Citado 2 vezes nas páginas 54 e 71.

AVRbeginners.net. **Timers**. 2017. Disponível em: <http://www.avrbeginners.net/architecture/timers/timers.html>. Citado na página 55.

BANERJEE, A. et al. Ensuring safety, security, and sustainability of mission-critical cyber–physical systems. **Proceedings of the IEEE**, IEEE, v. 100, n. 1, p. 283–299, 2012. Citado na página 15.

BEZ, R. et al. Introduction to flash memory. **Proceedings of the IEEE**, IEEE, v. 91, n. 4, p. 489–502, 2003. Citado na página 20.

BIOLCHINI, J. et al. Systematic review in software engineering. **System Engineering and Computer Science Department COPPE/UFRJ, Technical Report ES**, v. 679, n. 05, p. 45, 2005. Citado na página 30.

BOCCARDO, D. R. et al. Traceability of executable codes using neural networks. In: BURMESTER, M. et al. (Ed.). **International Conference on Information Security**. Springer Berlin Heidelberg, 2010. (Lecture Notes in Computer Science, v. 6531), p. 241–253. ISBN 978-3-642-18177-1. Disponível em: <http://dx.doi.org/10.1007/978-3-642-18178-8_21>. Citado na página 17.

CARMO, L.; MADRUGA, E. L.; MACHADO, R. C. S. Aspectos de Segurança da Informação em Redes de Medidores de energia Elétrica. In: UNIVERSIDADE FEDERAL DE CAMPINA GRANDE. **VIII International Seminar on Electrical Metrology**. João Pessoa, Brasil, 2009. Citado na página 21.

CARMO, L. F. R. d. C.; MACHADO, R. C. S. Verificação de integridade de software embarcado através de análise de tempo de resposta. In: **Anais do IX Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais**. Campinas – SP, Brasil: Sociedade Brasileira de Computação, 2009. Disponível em: <http://www.lbd.dcc.ufmg.br/colecoes/sbseg/2009/012.pdf>. Citado 5 vezes nas páginas 21, 35, 38, 40 e 71.

CASTELLUCCIA, C. et al. On the difficulty of software-based attestation of embedded devices. In: ACM. **Proceedings of the 16th ACM conference on Computer and communications security**. New York, New York, USA: ACM Press, 2009. p. 400–409. ISBN 9781605588940. Disponível em: <http://portal.acm.org/citation.cfm?doid=1653662.1653711>. Citado 5 vezes nas páginas 21, 27, 37, 38 e 73.

CASTRO, C. G. D. et al. Fvis : Uma ferramenta de ensaio de integridade de software. In: INSTITUTO NACIONAL DE TECNOLOGíA INDUSTRIAL. **Proceedings of The 10th Internacional Congress on Electrical Metrology**. Buenos Aires, 2013. Citado na página 26.

CASTRO, C. G. de et al. Evinced: Integrity verification scheme for embedded systems based on time and clock cycles. In: **The 15th IEEE International Conference on Pervasive Intelligence and Computing [Picom-2017]**. [S.l.: s.n.], 2017. Submitted. Citado na página 73.

Cathedrow. **Cryptosuite**. 2010. Disponível em: <https://github.com/Cathedrow/Cryptosuite>. Citado na página 57.

CHUI, M.; LÖFFLER, M.; ROBERTS, R. The internet of things. **McKinsey Quarterly**, n. Number 2, 2010. Citado na página 15.

COLLBERG, C.; NAGRA, J. **Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection**. 1. ed. Rio de Janeiro: Editora Ciência Modena Ltda., 2010. ISBN 978-85-7393-911-8. Citado na página 21.

COSTAN, V.; DEVADAS, S. Intel sgx explained. **IACR Cryptology ePrint Archive**, v. 2016, p. 86, 2016. Citado na página 74.

DE, P.; LIU, Y.; DAS, S. K. An epidemic theoretic framework for vulnerability analysis of broadcast protocols in wireless sensor networks. **IEEE Transactions on Mobile Computing**, IEEE, v. 8, n. 3, p. 413–425, 2009. Citado na página 20.

DENG, J.; HAN, R.; MISHRA, S. Secure code distribution in dynamically programmable wireless sensor networks. In: IEEE. **2006 5th International Conference on Information Processing in Sensor Networks**. Nashville, TN, USA, 2006. p. 292–300. Citado na página 15.

EASTLAKE 3RD, D.; JONES, P. **US Secure Hash Algorithm 1 (SHA1)**. IETF, 2001. RFC 3174 (Informational). (Request for Comments, 3174). Updated by RFCs 4634, 6234. Disponível em: <http://www.ietf.org/rfc/rfc3174.txt>. Citado na página 70.

ELDEFRAWY, K. et al. SMART: secure and minimal architecture for (establishing dynamic) root of trust. In: **19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012**. [s.n.], 2012. Disponível em: <http://www.internetsociety.org/smart-secure-and-minimal-architecture-establishing-dynamic-root-trust>. Citado 2 vezes nas páginas 21 e 36.

EVANS, B. **Beginning Arduino Programming**. Apress, 2011. (Apresspod Series). ISBN 9781430237778. Disponível em: <https://books.google.com.br/books?id=OyF7FK6OSekC>. Citado na página 54.

EVANS, C. C. **YAML 1.2**. 2017. Disponível em: <http://www.yaml.org/>. Citado na página 33.

EVERITT, B.; SKRONDAL, A. **The Cambridge dictionary of statistics**. New York: Cambridge University Press, 2002. Citado 2 vezes nas páginas 61 e 67.

FERRANTE, M.; SALTALAMACCHIA, M. The coupon collector's problem. In: UNIVERSITAT AUTòNOMA DE BARCELONA. **Materials matemàtics**. 2014. p. 01–35. Disponível em: <https://ddd.uab.cat/record/132177>. Citado na página 48.

FRANCILLON, A. et al. A minimalist approach to remote attestation. In: **Proceedings of the Conference on Design, Automation & Test in Europe**. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2014. (DATE '14), p. 244:1–244:6. ISBN 978-3-9815370-2-4. Disponível em: <http://dl.acm.org/citation.cfm?id=2616606.2616905>. Citado 3 vezes nas páginas 22, 34 e 38.

Gnu. **GNU C Library master sources**. 2017. Disponível em: <http://sourceware.org/git/?p=glibc.git;a=blob;f=stdlib/random_r.c;hb=glibc-2.15#l361>. Citado na página 59.

HEARST, M. A. et al. Support vector machines. **IEEE Intelligent Systems and their applications**, IEEE, v. 13, n. 4, p. 18–28, 1998. Citado na página 73.

HORSCH, J. et al. Sobtra: A software-based trust anchor for arm cortex application processors. In: **Proceedings of the 4th ACM Conference on Data and Application Security and Privacy**. New York, NY, USA: ACM, 2014. (CODASPY '14), p. 273–280. ISBN 978-1-4503-2278-2. Disponível em: <http://doi.acm.org/10.1145/2557547.2557569>. Citado 2 vezes nas páginas 35 e 38.

HOWARTH, P. et al. **Metrology–in short**. 3rd. ed. EURAMET, 2008. Disponível em: <https://www.euramet.org/Media/docs/Publications/Metrology_in_short_3rd_ed.pdf>. Citado na página 16.

IEEE. **IEEE Xplore Digital Library**. 2017. Disponível em: <http://ieeexplore.ieee.org/search/advsearch.jsp?expression-builder>. Citado na página 32.

IEEE, T.; GROUP, T. O. **diff - compare two files**. 2016. Disponível em: <http://pubs.opengroup.org/onlinepubs/9699919799/utilities/diff.html>. Citado na página 29.

INMETRO. **Vocabulário Internacional de Termos de Metrologia Legal (VIML)**. Inmetro, 2016. Disponível em: <http://www.inmetro.gov.br/legislacao/rtac/pdf/RTAC002399.pdf>. Citado 2 vezes nas páginas 16 e 17.

Intel Corporation. **Intel® 64 and IA-32 Architectures Software Developer's Manual – Volume 2B: Instruction Set Reference, M-U**. 2016. Disponível em: <https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-2b-manual.pdf>. Citado na página 40.

ISMAIL, R.; SYED, T. A.; MUSA, S. Design and implementation of an efficient framework for behaviour attestation using n-call slides. In: **Proceedings of the 8th International Conference on Ubiquitous Information Management and Communication**. New York, NY, USA: ACM, 2014. (ICUIMC '14), p. 36:1–36:8. ISBN 978-1-4503-2644-5. Disponível em: <http://doi.acm.org/10.1145/2557977.2558002>. Citado 3 vezes nas páginas 35, 37 e 38.

ISO.ORG. **Information technology – Trusted platform module library – Part 1: Architecture**. 2016. Disponível em: <https://www.iso.org/standard/66510.html>. Citado na página 36.

JABREF. **Jabref**. 2017. Disponível em: <http://www.jabref.org/>. Citado na página 34.

JCGM 200:2012. **Vocabulário Internacional de Metrologia: conceitos fundamentais e gerais de termos associados (VIM 2012)**. 3. ed. Duque de Caxias, RJ: INMETRO, 2012. 94 p. ISBN 9788586920097. Citado na página 16.

KITCHENHAM, B. **Procedures for performing systematic reviews**. Software Engineering Group, Department of Computer Science, Keele University, Keele, Staffs, ST5 5BG, UK, 2004. Citado na página 30.

KITCHENHAM, B. et al. Systematic literature reviews in software engineering–a tertiary study. **Information and Software Technology**, Elsevier, v. 52, n. 8, p. 792–805, ago. 2010. ISSN 09505849. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0950584910000467http://linkinghub.elsevier.com/retrieve/pii/S0950584910000467>. Citado na página 30.

KITCHENHAM, B. A.; CHARTERS, S. **Guidelines for performing Systematic Literature Reviews in Software Engineering**. Software Engineering Group, School of Computer Science and Mathematics, Keele University, Keele, Staffs, ST5 5BG, UK, 2007. Citado na página 30.

KLIR, G.; YUAN, B. **Fuzzy sets and fuzzy logic**. New Jersey: Prentice hall, 1995. Citado na página 73.

KOCABAS, U. et al. Poster: Practical embedded remote attestation using physically unclonable functions. In: **Proceedings of the 18th ACM Conference on Computer and Communications Security**. New York, NY, USA: ACM, 2011. (CCS '11), p. 797–800. ISBN 978-1-4503-0948-6. Disponível em: <http://doi.acm.org/10.1145/2046707.2093496>. Citado 2 vezes nas páginas 36 e 38.

KONG, J. et al. Pufatt: Embedded platform attestation based on novel processor-based pufs. In: **Proceedings of the 51st Annual Design Automation Conference**. New York, NY, USA: ACM, 2014. (DAC '14), p. 109:1–109:6. ISBN 978-1-4503-2730-5. Disponível em: <http://doi.acm.org/10.1145/2593069.2593192>. Citado 2 vezes nas páginas 36 e 38.

KOVAH, X. et al. New results for timing-based attestation. In: **2012 IEEE Symposium on Security and Privacy**. San Francisco, CA, USA: IEEE, 2012. p. 239–253. ISSN 1081-6011. Disponível em: <http://ieeexplore.ieee.org/abstract/document/6234416/>. Citado na página 36.

LANGHEINRICH, M. Privacy by design—principles of privacy-aware ubiquitous systems. In: **Ubicomp 2001: Ubiquitous Computing**. Berlin, Heidelberg: Springer, 2001. p. 273–291. Disponível em: <https://link.springer.com/book/10.1007/3-540-45427-6>. Citado na página 15.

LEITãO, F. d. O.; VASCONCELLOS, M. T.; BRANDãO, P. C. R. Legal metrology actions to avoid volume frauds on fuel dispensers. In: Instituto Nacional de Metrologia, Qualidade e Tecnologia, Sociedade Brasileira de Metrologia. **Anais do 3o. Congresso Internacional de Metrologia Mecânica**. Gramado–RS, Brasil, 2014. Citado na página 17.

LI, Y.; MCCUNE, J. M.; PERRIG, A. Viper: Verifying the integrity of peripherals' firmware. In: **Proceedings of the 18th ACM Conference on Computer and Communications Security**. New York, NY, USA: ACM, 2011. (CCS '11), p. 3–16. ISBN 978-1-4503-0948-6. Disponível em: <http://doi.acm.org/10.1145/2046707.2046711>. Citado 3 vezes nas páginas 35, 38 e 70.

MACHADO, R. C. S. et al. **Validação de software em Metrologia Legal**. 2010. Disponível em: <http://xrepo01s.inmetro.gov.br/handle/10926/588>. Citado na página 17.

MAES, R. **Physically Unclonable Functions**. Berlin, Heidelberg: Springer, 2013. Citado na página 36.

NIST. Secure hash standard (shs). **FIPS PUB 180-4**, v. 4, 2012. Disponível em: <http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf>. Citado na página 57.

OIML. **OIML D 1 Considerations for a Law on Metrology**. Paris, France, 2012. Disponível em: <https://www.oiml.org/en/files/pdf_d/d001-e12.pdf>. Citado na página 16.

PADMAVATHI, D. G.; SHANMUGAPRIYA, M. et al. A survey of attacks, security mechanisms and challenges in wireless sensor networks. **International Journal of Computer Science and Information Security, IJCSIS**, USA, v. 4, n. 1 and 2, August 2009. Disponível em: <http://arxiv.org/abs/0909.0576>. Citado na página 20.

PARK, S. K.; MILLER, K. W. Random number generators: good ones are hard to find. **Communications of the ACM**, ACM, v. 31, n. 10, p. 1192–1201, 1988. Citado na página 59.

PEDREGOSA, F. et al. Scikit-learn: Machine learning in Python. **Journal of Machine Learning Research**, v. 12, p. 2825–2830, 2011. Citado na página 66.

PICom-2017 Web Team. **The 15th IEEE International Conference on Pervasive Intelligence and Computing PICOM-22017**. 2017. Disponível em: <http://cse.stfx.ca/~picom2017/>. Citado na página 73.

PRADO, C. B. D. et al. Software analysis and protection for smart metering. **NCSLI Measure**, Taylor & Francis, v. 9, n. 3, p. 22–29, 2014. Disponível em: <http://www.tandfonline.com/doi/abs/10.1080/19315775.2014.11721691>. Citado 4 vezes nas páginas 18, 23, 25 e 28.

PRESCHERN, C. et al. Software-based remote attestation for safety-critical systems. In: **Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on**. Luxembourg, Luxembourg: IEEE, 2013. p. 8–12. Disponível em: <http://ieeexplore.ieee.org/abstract/document/6571600/>. Citado 2 vezes nas páginas 36 e 38.

Python Software Foundation. **Python 3.5**. 2017. Disponível em: <https://docs.python.org/3.5/reference/>. Citado 3 vezes nas páginas 54, 55 e 72.

RAUTER, T. et al. Privilege-based remote attestation: Towards integrity assurance for lightweight clients. In: **Proceedings of the 1st ACM Workshop on IoT Privacy, Trust, and Security**. New York, NY, USA: ACM, 2015. (IoTPTS '15), p. 3–9. ISBN 978-1-4503-3449-5. Disponível em: <http://doi.acm.org/10.1145/2732209.2732211>. Citado na página 36.

RENNA, R. B. D. et al. **Introdução ao kit de desenvolvimento Arduino**. 2013. Versão A2013M10D02. Disponível em: <http://www.telecom.uff.br/pet/petws/downloads/tutoriais/arduino/Tut_Arduino.pdf>. Citado na página 54.

SCHULZ, S.; SADEGHI, A.-R.; WACHSMANN, C. Short paper: Lightweight remote attestation using physical functions. In: **Proceedings of the Fourth ACM Conference on Wireless Network Security**. New York, NY, USA: ACM, 2011. (WiSec '11), p. 109–114. ISBN 978-1-4503-0692-8. Disponível em: <http://doi.acm.org/10.1145/1998412.1998432>. Citado 2 vezes nas páginas 36 e 38.

Scikit-learn Developers. **RobustScaler**. 2017. Disponível em: <http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.RobustScaler.html>. Citado na página 66.

SESHADRI, A.; LUK, M.; SHI, E. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. **ACM SIGOPS Operating Systems Review**, v. 39, n. 5, p. 1–16, 2005. Disponível em: <http://dl.acm.org/citation.cfm?id=1095812>. Citado 6 vezes nas páginas 26, 35, 38, 48, 53 e 70.

SESHADRI, A. et al. Swatt: software-based attestation for embedded devices. In: **IEEE Symposium on Security and Privacy, 2004, Proceedings**. Berkeley, CA, USA, USA: IEEE, 2004. p. 272–282. ISSN 1081-6011. Citado 13 vezes nas páginas 21, 22, 25, 26, 29, 34, 35, 37, 38, 45, 48, 53 e 70.

Sinmetro; Conmetro; CBR. **Guia de Boas Práticas de Regulamentação**. [S.l.], 2015. Disponível em: <http://www.inmetro.gov.br/qualidade/pdf/guia_portugues.pdf>. Citado na página 16.

SONDOW, J.; WEISSTEIN, E. W. **Harmonic Number.** 2017. Disponível em: <http://mathworld.wolfram.com/HarmonicNumber.html>. Citado na página 60.

SORBER, J. M. et al. Plug-n-trust: Practical trusted sensing for mhealth. In: **Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services**. New York, NY, USA: ACM, 2012. (MobiSys '12), p. 309–322. ISBN 978-1-4503-1301-8. Disponível em: <http://doi.acm.org/10.1145/2307636.2307665>. Citado na página 37.

SPINELLIS, D. Reflection as a mechanism for software integrity verification. **ACM Transactions on Information and System Security**, v. 3, n. 1, p. 51–62, fev. 2000. ISSN 10949224. Disponível em: <http://dl.acm.org/citation.cfm?id=353323.353383>. Citado 8 vezes nas páginas 20, 21, 26, 27, 29, 34, 38 e 70.

SRINIVASAN, R.; DASGUPTA, P.; GOHAD, T. Software based remote attestation for os kernel and user applications. In: **Privacy, Security, Risk and Trust (PASSAT) and 2011 IEEE Third Inernational Conference on Social Computing (SocialCom), 2011 IEEE Third International Conference on**. Boston, MA, USA: IEEE, 2011. p. 1048–1055. Citado 2 vezes nas páginas 36 e 38.

STALLINGS, W. **Cryptography and Network Security: Principles and Practice**. 6. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2013. ISBN 0133354695, 9780133354690. Citado 2 vezes nas páginas 24 e 25.

WEISSTEIN, E. W. **Euler-Mascheroni Constant**. 2017. Disponível em: <http://mathworld.wolfram.com/Euler-MascheroniConstant.html>. Citado na página 60.

Welmec. **Welmec 1 WELMEC – An Introduction**. Thijsseweg 11, NL – 2629 JA Delft, The Netherlands, 2011. Disponível em: <http://www.welmec.org/latest/guides/1/>. Citado na página 18.

Welmec. **Welmec 7.2 Software Guide (Measuring Instruments Directive 2004/22/EC)**. Thijsseweg 11, NL – 2629 JA Delft, The Netherlands, 2012. Disponível em: <http://www.welmec.org/latest/guides/72/>. Citado na página 18.

WOLCHOK, S. et al. Security analysis of india's electronic voting machines. In: ACM. **Proceedings of the 17th ACM Conference on Computer and Communications Security**. New York, NY, USA: ACM, 2010. (CCS '10), p. 1–14. ISBN 978-1-4503-0245-6. Disponível em: <http://doi.acm.org/10.1145/1866307.1866309>. Citado na página 15.

YAN, Q. et al. A software-based root-of-trust primitive on multicore platforms. In: **Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security**. New York, NY, USA: ACM, 2011. (ASIACCS '11), p. 334–343. ISBN 978-1-4503-0564-8. Disponível em: <http://doi.acm.org/10.1145/1966913.1966957>. Citado 2 vezes nas páginas 36 e 38.

ZHAO, S. et al. Providing root of trust for arm trustzone using on-chip sram. In: **Proceedings of the 4th International Workshop on Trustworthy Embedded Devices**. New York, NY, USA: ACM, 2014. (TrustED '14), p. 25–36. ISBN 978-1-4503-3149-4. Disponível em: <http://doi.acm.org/10.1145/2666141.2666145>. Citado 2 vezes nas páginas 35 e 38.